# Detection, Categorization and Repair of Flaky Tests Using Large Language Models

by

Sakina Fatima

Thesis submitted to the
Faculty of Engineering
In partial fulfillment of the requirements
For the Doctorate of Philosophy degree in
Computer Science

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

**Abstract**

Software testing is critical for ensuring software dependability. However, some test cases, known as flaky tests, exhibit non-deterministic behavior, passing or failing inconsistently even with the same source code version. These flaky tests create significant overhead in software development, requiring developers to rerun tests or debug code unnecessarily. Traditional approaches for detecting flaky tests involve rerunning them multiple times, a process that is computationally expensive and impractical for large test suites. Machine learning (ML) models have been proposed as a scalable alternative to predict flaky tests without reruns. However, existing ML-based techniques often rely on production code or project-specific features, limiting their generalizability across diverse projects. Furthermore, the use of predefined feature sets results in suboptimal accuracy when applied to realistic datasets. To address these challenges, we propose two novel, black-box, large language model-based solutions: (a) Flakify: A flaky test predictor that relies solely on the source code of test cases, eliminating the need for access to production code or predefined feature sets. Flakify uses CodeBERT, a pre-trained language model, and demonstrates superior performance on two benchmark datasets. It achieves F1-scores of 79% and 73% using cross-validation and per-project validation on the FlakeFlagger dataset, and 98% and 89% on the IDoFT dataset. Flakify outperforms the state-of-the-art solution (FlakeFlagger) by 10 and 18 percentage points in precision and recall, respectively. (b) FlakyFix: A framework designed to predict the required fix for flaky tests by classifying them into 13 distinct fix categories based solely on test code analysis. By leveraging code models and few-shot learning, FlakyFix accurately predicts most fix categories. To further enhance flaky test repairs, we augment GPT 3.5 Turbo prompts with predicted fix category labels. Our experimental results show that 51% to 83% of GPT-suggested repairs pass, with only 16% of the test code needing further modifications for the remaining cases. These two approaches significantly reduce the overhead associated with rerunning flaky tests and provide an efficient method for predicting and repairing flaky tests, making them more suitable for real-world industrial applications.

**Acknowledgements**

I would like to express my deep gratitude to Huawei Technologies for their financial support and insightful feedback throughout this research. This work was also supported by funding from Mitacs Canada, the Canada Research Chair, and the Discovery Grant programs of the Natural Sciences and Engineering Research Council of Canada (NSERC), as well as by the Science Foundation Ireland grant 13/RC/2094-2 and Alberta Innovates.

I am especially thankful to Dr. Taher A. Ghaleb and Dr. Hadi Hemmati, whose guidance was invaluable, particularly during the more challenging phases of my research. The experiments in this work were made possible, in part, by the resources provided by WestGrid (`https://www.westgrid.ca`) and Compute Canada (`https://www.computecanada.ca`).

I also extend my sincere thanks to the members of my Thesis Advisory Committee, Dr. Guy-Vincent Jourdan and Dr. Nafiseh Kahani, whose valuable feedback and ongoing support have significantly improved this thesis.

Finally, I am profoundly grateful to my supervisor, Dr. Lionel Briand, for their unwavering support and patience throughout this journey. Their critical insights and constructive feedback were pivotal to the successful completion of this work.

## Dedication

*"Verily, with every hardship comes ease."* – (Qur'an 94:6)

All praise is due to Allah Almighty, whose infinite wisdom and guidance granted me the strength to complete this challenging journey. This milestone is the result of the unwavering support and prayers of my loved ones, to whom I am deeply indebted.

To my mother, the pillar of my strength, this PhD is as much your achievement as it is mine. It was your dream, your belief in me, and your endless encouragement that kept me moving forward. Your long calls, where you patiently listened to my struggles and celebrated my small victories, were my lifeline. From staying awake until I was safely home to remembering details about my research, your love and support have been my anchor.

To my father, whose prayers and motivational words have been a constant source of guidance and strength, thank you for giving me the freedom to pursue my dreams at world-renowned institutions.

To my sister and my niece, thank you for bringing joy and laughter into my life, even from afar, and for being my much-needed happy distraction.

To my husband, Shan-E-Abbas, and his wonderful family, meeting you halfway through this journey was a blessing. Your love, encouragement, and unwavering belief in me made the challenges more bearable.

To my friends: Sara, Duaa, Fatemah, Chaima, Gunay, Kainat, Rahatara, Tayyaba, Gul, Saurav, and my colleagues at Nanda Lab, thank you for your heartfelt wishes, endless encouragement, support, and constructive feedback. Your presence has been a gift.

Lastly, I dedicate this work to my younger self—the girl who dared to dream big, and refused to give up, even when the weight of it all felt overwhelming.

Here's to new beginnings, endless opportunities, and always dreaming big.

# Table of Contents

# List of Tables

# List of Figures

# Glossary of Acronyms

**AST** Abstract Syntax Tree

**BLEU** Bilingual Evaluation Understudy Score

**FNN** Feedforward Neural Network

**FP** False Positives

**FSL** Few Shot Learning

**Flakify** Flaky Tests Classification

**FlakyFix** Flaky Tests Fix

**FN** False Negatives

**GPT** Generative Pre-trained Transformer

**LLM** Large Language Model

**LM** Language Model

**LR** Logistic Regression

**ML** Machine Learning

**RQ** Research Question

**TN** True Negatives

**TP** True Positives

# Chapter 1

# Introduction

## 1.1 Motivation

Software testing is an essential activity to assure software dependability. When a test case fails, it usually indicates that recent code changes were incorrect. However, it has been observed, in many environments, that test cases can be non-deterministic, passing and failing across executions, even for the same version of the source code. These test cases are referred to as flaky test cases [21, 60, 131]. Such test cases can introduce overhead to software development, since they require developers to either (a) debug the production or testing code looking for a bug that might not exist, or (b) rerun a failed test case multiple times to check if it would eventually pass, thus suggesting that the failure is not due to recent code changes but to the test case itself.

Previous research has investigated the common reasons behind test flakiness, such as concurrency, resource leakage, and test smells. The conventional approach to detect flaky test cases is to rerun them numerous times [10, 53], which is in most practical cases computationally expensive [71] or even impossible. To address this issue, recent studies have proposed approaches using machine learning (ML) models to predict flaky test cases without rerunning them [6, 13, 93], thus proposing a much more scalable and practical solution. Despite significant progress, these approaches (a) rely on production code, which is not always accessible by software test engineers or a scalable solution, or (b) employ project-specific features as flaky test case predictors, which makes them inapplicable to other projects. Moreover, these approaches rely on a limited set of pre-defined features, extracted from the source code of test cases and the system under test. However, when evaluated on realistic datasets, these approaches yield a relatively low accuracy (F1-scores in the range 19%-66%), thus suggesting they may not capture enough information about test flakiness. Finding additional features that could potentially be associated with flaky test cases, preferably based on test code only (black-box), is therefore a research challenge.

Secondly, once these flaky tests are detected, developers must repair them. The process involves first localizing the code causing the issue, whether in the test code or the production code, and then addressing the flakiness in the specific test method.

We propose two black-box (using test code only) solutions using large language models: (a) Flakify, which analyzes test code using a pre-trained LM to predict flakiness without rerunning the tests, and (b) FlakyFix, which applies the LLM to categorize the type of fix needed for flaky tests and then generates a full or partial repair suggestion. These models are trained to understand and process code structure, syntax, and patterns associated with flaky tests, either through pre-training or additional fine-tuning with examples. Flakiness in tests can stem from highly diverse reasons [39] and can be resolved through various strategies, as evident from the International Dataset of Flaky Tests (IDoFT) [50], the largest open-source repository containing Java and Python flaky tests and their respective fixes. Remedies range from altering the test execution order to addressing operating system or implementation dependencies by code modifications in the test class. Our study mainly focuses on repairing flaky tests in a black-box manner (no access to production code) by updating flaky statements within the test case code, which constitutes 10% (562 tests from accepted pull requests) of the total 5,500 tests in the Java dataset [50]. While such flaky tests represent a minority in this dataset, techniques to fix them are warranted, thereby serving the important needs of the testing community in charge of writing and evolving the test code. In addition, studies such as that of Akli et al. [4] report that, in their dataset, 70% of flakiness issues originate in the test code [52, 60], further supporting our focus. The distinction between fixing the production code and the test code is crucial.

## 1.2 Thesis Objectives

The long-term goal of this research is to detect flaky tests without rerunning them, to categorize the flaky tests based on their fix categories, and then finally repair them automatically. This is refined into the following three main objectives:

**$TO_1$ (Detection of Flaky Tests):** Predict flaky tests using LM.

**$TO_2$ (Categorization of Flaky Tests):** Predict the category of flaky tests according to the different types of fixes required to repair these tests using different LMs.

**$TO_3$ (Repair of Flaky Tests):** Based on the predicted category, repair flaky test code using LLMs.

## 1.3 Contributions

The research contributions of this thesis include:

1. Flakify: A generic, black-box, language model-based flaky test case predictor, which does not require rerunning test cases. It predicts flaky test cases on the basis of their code without requiring the definition of features. This was the first time a LM is used for predicting flaky tests.

2. An Abstract Syntax Tree (AST)-based technique for statically detecting and only retaining statements that match eight test smells in the test code, thus enhancing the application of LM in Flakify.

3. Running a large scale experiment of predicting flaky tests on two distinct datasets: FlakeFlagger containing 21,661 test cases collected from 23 Java projects, and the International Dataset of Flaky Tests (IDoFT) containing 3,862 test cases collected from 312 Java projects.

4. Definition of a categorization for flaky test fixes. The goal is to provide practical guidance, to a LLM or human, for fixing tests.

5. Development of a set of heuristics (search rules) and accompanying open-source scripts to automatically label flaky tests based on their fixes, along with the creation of a public dataset of labeled flaky tests for training, using our automated script.

6. A novel black box LLM based framework called FlakyFix that:

   (a) Predicts flaky test fix categories using code models (LM that are pre-trained on code) and few-shot learning (FSL).

   (b) Automatically generates fixes for flaky tests using a LLM (GPT 3.5 Turbo), with the predicted fix categories.

Contributions 1–3 address $TO_1$, contributions 4–5 address $TO_2$ whereas contributions 6 address $TO_3$. While the author of this thesis is the main author and contributor to all of the above contributions, contributions 1–3 benefited from the co-supervision of Dr. Taher A Ghaleb in terms of refining the idea, code contributions, empirical evaluation design and writing. Similarly, the work done related to contributions 4–6 benefited from Dr. Hadi Hemmati's guidance in terms of idea, empirical evaluation design, writing. It goes without saying that all the contributions above were supervised, guided, and reviewed by the supervisor of this thesis, Prof. Lionel Briand.

## 1.4 Publications

In addition to this thesis, the research results have been communicated to the public via the following publications:

1. **Sakina Fatima**, Taher A. Ghalib, Lionel C. Briand, "Flakify: A Black-Box, Language Model-Based Predictor for Flaky Tests," in *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1912 - 1927, April. 2023, doi: 10.1109/TSE.2022.3201209

2. **Sakina Fatima**, Taher A. Ghalib, Lionel C. Briand, "Journal-First Presentation: Flakify: A Black-Box, Language Model-Based Predictor for Flaky Tests," in *IEEE/ACM 45th International Conference on Software Engineering*, Journal-first track, Melbourne, Australia, IEEE/ACM, 2023.

3. **Sakina Fatima**, Hadi Hemmati, Lionel C. Briand, "FlakyFix: Using Large Language Models for Predicting Flaky Test Fix Categories and Test Code Repair," in *IEEE Transactions on Software Engineering*, October. 2024, doi: 10.1109/TSE.2024.3472476

The detailed contributions of each publication is listed in section 1.3, where contributions 1–4 correspond to publications 1 and 2; and contributions 5–8 correspond to publication 3. We have provided a detailed statement of contributions of each co-author of the publications in section 1.3.

## 1.5 Thesis Structure

The rest of this thesis is structured as follows:

- chapter 2 addresses $TO_1$ by providing:
  - a background about flaky tests and LM,
  - a detailed description of our black-box, LM-based approach for predicting flaky tests,
  - an evaluation of our approach, reporting experimental results and discussing their implications on the detection of flaky tests.
- chapter 3 addresses $TO_2$ by providing:
  - a detailed description of our approach for developing an automated tool for labeling flaky tests with fix categories and predicting such categories using code models, with and without few shot learning (FSL),
  - an evaluation of our approach to categorize the flaky tests, including experimental results and a discussion on their implications,
  - a qualitative analysis of different smaller code models to demonstrate and explain their ability to learn from test code and subsequently make predictions about categories of the fix required to repair the flaky test.
- chapter 4 addresses $TO_3$ by providing:
  - a detailed description of our approach for repairing flaky tests using LLMS, including prompt engineering and in-context learning,
  - an evaluation of our approach for repairing flaky tests, including experimental results and a discussion of their implications.
- Finally, chapter 5 discusses the conclusions, practical implications of this research, its limitations, and future work.

# Chapter 2

# Detection of Flaky Tests

This chapter addresses $TO_1$, i.e. black-box LM-based detection of flaky tests. The contents of this chapter have been published in the Journal of *Transactions on Software Engineering (TSE)* [27].

## 2.1 Overview

As discussed in chapter 1, *Chapter Structure*. The rest of this section is structured as follows. Section 2.2 provides background materials on the problem of flaky tests, challenges in detecting these tests, and the use of LMs. Section 2.3 discusses our flaky test detection approach in detail. Section 2.4 presents the evaluation of flaky test detection and discusses the results. Section 2.5 explains threats to the validity. Section 2.6 reports related work and finally, Section 2.7 concludes this chapter.

## 2.2 Background

In this section, we describe flaky tests, their root causes, their practical impact, and the strategies to detect them. In addition, we describe pre-trained LMs and how they can potentially contribute to predicting flaky test cases.

### 2.2.1 Flaky Test Cases

In software testing, a flaky test refers to test cases that intermittently fail and pass across executions, even for the same version of the source code, i.e., non-deterministically behaving test cases [131]. Flaky test cases lead to many problems during software testing, by producing unreliable results and wasting time and computational resources. A flaky test can also fail for different reasons across executions, making it difficult to identify which failures are actually related to faults in the system under test.

Flaky test cases have been reported to be a significant problem in practice at many companies including Google, Huawei, Microsoft, SAP, Spotify, Mozilla, and Facebook [9, 38,51,130]. As reported by Google, almost 16% of their 4.2 million test cases are flaky [71]. Microsoft has also reported that 26% of $3.8k$ build failures were due to flaky test cases. Many studies have been conducted to study flaky test cases, their causes, and the solutions to address them [6, 10, 13, 60, 85, 93, 94, 131]. Prominent causes of flaky test cases include asynchronous waits, test order dependency, concurrency, resource leakage, and incorrect test inputs or outputs. In addition, flaky test cases were found to be associated with other factors, such as test smells, which are further discussed below.

### 2.2.2 Flaky Test Case Detection

The most common approach for detecting flaky test cases is by rerunning test cases numerous times to check whether they behave consistently across executions [10,53]. Though effective, this approach is computationally expensive and not practical in many situations, for example in continuous integration contexts, where builds are submitted automatically and frequently to perform regression testing. To mitigate such cost, other approaches attempted to detect flaky test cases without relying on rerunning them. To that end, characteristics of test cases, such as execution history, coverage information, and static test features, were used to predict whether a test case is flaky or not. Prediction models were built using ML and Natural Language Processing (NLP) techniques [6, 13, 93]. Such techniques require training ML models with pre-defined sets of features used as indicators for test flakiness. Such features commonly present practical limitations, such as (a) their reliance on production code, which is not always accessible or efficiently analyzable by test engineers, and (b) their limited capacity to capture the actual structure or behavior of test cases, such as the use of language keywords [93] or the presence of test smells [6, 13, 94] in test code.

After identifying potentially flaky test cases, developers can focus their investigation on them and, hence, attempt to fix code statements causing such flakiness. Developers may also choose to rerun those specific test cases many more times to verify that they are actually flaky [86]. This is a reasonable undertaking, since test cases predicted as flaky normally represent a small percentage of the entire test suite. This, in turn, significantly eliminates a large part of the effort and time required to investigate or rerun test cases whenever a failure occurs [6].

### 2.2.3 Test Smells

Test Smells are inappropriate design or implementation choices made by developers while writing test cases [109]. Though test smells might not harm the functionality of a test case, previous research has reported that they tend to be associated with test flakiness. Test smells were further employed to classify whether a test case is flaky or not. For example, test smells in Table 2.1 were part of the features used by Alshammari et al. [6] to predict test flakiness. Camara et al. [13] also used a more comprehensive set of test smells for flaky

| Test Smell | Description |
|---|---|
| Indirect Testing | A test interacts with the class under test using methods from other classes |
| Eager Testing | A test performs multiple checks for various functionalities |
| Test Run War | A test allocates files or resources that might be used by other test cases |
| Conditional Logic | A test uses a conditional *if* statement |
| Fire and Forget | A test launches background threads or processes |
| Mystery Guest | A test accesses external resources |
| Assertion Roulette | A test performs multiple assertions |
| Resource Optimism | A test accesses external resources without checking their existence |

Table 2.1: Test smells used by FlakeFlagger [6]

test case prediction. Results showed that *Sleepy Test* and *Assertion Roulette* are among test smells that are highly associated with flaky test cases.

### 2.2.4 Pre-trained Language Models

Much research has been carried out in the field of NLP for developing pre-trained LMs.LMs estimate the probability of different linguistic units, i.e., words, symbols, and sequence of them, occurring in a given sentence. There are many LMs proposed in the literature, such as BERT [18], ELMo [92], XLNet [123], RoBERTa [59], and VideoBERT [106]. These models were pre-trained, using self-supervised learning, on a large corpus of unlabelled data. For example, BERT was pre-trained using a large dataset of English text collected from books and Wikipedia, whereas VideoBERT was pre-trained using a large dataset of instructional videos collected from YouTube.

Pre-trained LMs are often further fine-tuned using a specific, labelled dataset to train neural networks for performing various NLP tasks, such as text classification and entity recognition [74], relation extraction [8], sentence tagging, or next sentence prediction [18]. For example, BERT was fine-tuned to perform sentiment analysis [107, 122], trained on labelled datasets to assign sentiment tags, i.e., positive, negative, or neutral, to a given text. Fine-tuning requires initializing a LM with the same parameters used for pre-training, and then further training the model using labeled data related to a specific task.

LMs usually employ multi-layer transformers as a model architecture to perform many computations in parallel [110]. Transformer models adopt positional embedding to vectorize individual words by considering their positions in a given sequence of words. Thus, unlike Recurrent Neural Networks (RNNS) [61] and Long-Short Term Memory (LSTM) [41], transformer models do not require looking at past hidden states to capture dependencies with previous words in a sequence of words.

Given the wide popularity of LMs in various NLP applications, researchers have attempted to apply these LMs to programming languages. However, when BERT, for example, was used for detecting the architectural tactics in source code [47], e.g., recognizing software design patterns, the results were relatively worse compared to those obtained when BERT was used for natural language text. To address this issue, recent work proposed pre-training LMs on source code written in many programming languages in addition to

natural language text [29,36,45,46]. These models are well suited for fine-tuning to perform tasks related to source code. CodeBERT [29] is an example of a LM that was pre-trained on both natural and programming languages.

### CodeBERT

CodeBERT [29] is a LM that was pre-trained on a large, unlabeled dataset containing English text as well as source code written in six different programming languages, namely Java, JavaScript, Python, Ruby, PHP, and Go, obtained from the CodeSearchNet corpus [44]. CodeBERT takes, as input, source code statements and natural language sentences, which are then tokenized using the WordPiece [119] tokenizer. Similar to BERT and RoBERTa, CodeBERT uses a multi-layer bidirectional transformer [110] as model architecture. This transformer is composed of six layers, each of which contains 12 self-attention heads capturing word relationships, a hidden state, and a 768-dimensional vector, as the output of each layer.

CodeBERT also employed Masked Language Modeling (MLM) [18] and Replaced Token Detection (RTD) [16] during pre-training, allowing to take tokens from random positions and masking them with special tokens, which are later used to predict the original tokens. As a result, each token is assigned a vector representation containing information about the token and its position in a given code. The final output of CodeBERT is a single vector representation aggregating all individual vector representations. This vector representation can further be fine-tuned to perform various tasks, e.g., classification. For example, to evaluate the performance of CodeBERT, it was fine-tuned to perform two tasks: (1) code search, i.e., retrieving the most relevant code to a given natural language text; (2) code documentation, i.e., generating a natural language description for a given source code. Moreover, CodeBERT was also adopted to perform classification tasks, such as bug prediction [82] and vulnerability detection [118].

### Other Models for Programming Languages

As mentioned above, recently, many LMs for programming languages were proposed. For example, GraphCodeBERT [36] was pre-trained on the inherent structure of source code and its data flow showing variables dependencies. Similar to CodeBERT, GraphCodeBERT was used for code search, in addition to code translation and refinement as well as clone detection. Another model for programming languages is TreeBERT [45], which was pre-trained using AST representations of Java and Python source code. TreeBERT was used for code documentation, similar to CodeBERT, in addition to code summarization. There is also CuBERT [46], a programming LM pre-trained using Python source code. CuBERT was used for classification tasks, such as classifying exceptions and variable misuses.

Despite the capabilities of these models, CodeBERT has been the most commonly used LM and we selected it to address our objectives for several reasons presented below.

- The pre-trained CodeBERT model is publicly available.[1]

[1] `https://huggingface.co/microsoft/CodeBERT-base`

- Unlike GraphCodeBERT, CodeBERT does not take into consideration the data flow in a given source code, which might not be easy to capture using test code only. For example, unlike local variables, if a global or external variable is used by a test case, GraphCodeBERT cannot identify the type and value of that variable when analyzing test code only.
- Unlike TreeBERT, which requires converting source code into ASTs, CodeBERT only requires source code as input.
- Unlike CuBERT, which was only pre-trained on Python source code without comments, CodeBERT was pre-trained on multiple programming languages using both source code and natural language comments.

## 2.3 Black-Box Flaky Test Case Predictor

This section describes our black-box solution for predicting flaky test cases. This is motivated by making such predictions scalable, as white-box analysis of the production source code, especially in the context of large systems, is often not a viable solution.

### 2.3.1 CodeBERT for Flaky Test Case Prediction

In this chapter, we propose Flakify, a black-box solution for predicting whether a test case is flaky or not. Flakify relies solely on the source code of a test case and does not require to rerun it multiple times. The source code of test cases, i.e., Java test methods, includes the method declaration, body, and it associated Javadoc comments. While several studies have proposed ML techniques to predict flaky test cases, such techniques rely on pre-defined features extracted not only from the source code of test cases but also that of the system under test. However, results [6, 13, 93] suggest those features may not be enough, and finding additional features that could potentially be associated with flaky test cases remains a research challenge given their non-deterministic behavior. Therefore, we employed CodeBERT, the pre-trained code model described above, to perform a binary classification of test cases as *Flaky* or *Non-Flaky*. CodeBERT does not require to define features as it automatically identifies patterns based on the syntax and semantics of a given test code.

CodeBERT starts by converting the source code of a test case into a list of tokens, each of which is converted into an integer vector representation. Finally, an aggregated vector representation is generated as an output of CodeBERT, which is further fine-tuned to classify test cases as *Flaky* or *Non-Flaky*. Figure 2.1 presents an example of how the source code of a test case is converted into tokens and then into integer vector representations.

#### Source Code Tokenization

To transform the source code into tokens, the source code of test cases is tokenized by the WordPiece [119] tokenizer using a pre-generated vocabulary file containing the vocabulary

of both English and programming languages used for model pre-training. However, uncommon words, i.e., those that do not exist in the vocabulary file, are separated into several sub-words. For example, the CodeBERT tokenizer splits '`assertThat`' into '`assert`' and '`##that`', where '`##`' denotes that a token represents a sub-word. Then, if a token is not found in the vocabulary file, the unknown token, `<UNK>`, is used. For each input, two special tokens, `[CLS]` and `[SEP]`, are added. Eventually, for a given source code, the tokenizer generates a sequence of tokens in the form of `[CLS]`, $c_1, c_2, .., c_n$, `[SEP]`, where $c_i$ is a code token. The `[CLS]` token plays an important role in the classification of flaky test cases, as it contains the aggregated vector representation of all the vector representations of the tokens of a given test case. On the basis of that aggregated vector representation, our model classifies a test case as *Flaky* or *Non-Flaky*. `[SEP]` is just used to mark the end of the sequence of tokens. The tokenizer also adds '`Ġ`' in front of each word that is preceded by a whitespace in a statement.

### Converting Tokens into Vector Representations

Once the source code tokens are generated, each token, including sub-word, special, and unknown tokens, is mapped to an index, e.g., id 34603 for "Test" in Figure 2.1, based on the position and context of each word in a given input. Each token is described by an 768-dimensional integer vector generated during CodeBERT pre-training. Using token padding, the same token length is given to the code of all test cases used as input, e.g., "1" in Figure 2.1. However, CodeBERT has a limit of 512 tokens per input. As a result, any token sequence exceeding that limit is truncated, which might lead to removing code statements with potentially relevant information about test flakiness. In addition to *input ids* matching tokens, another list of *attention masks* is generated containing ones and zeros to help the model distinguish between code tokens, which should be given attention, and extra tokens added for padding. Finally, for each test case, token vectors are aggregated to form one vector characterizing the `[CLS]` token, which is also represented using a 768-sized vector referred to by the first input index '0'.

### Fine-Tuning CodeBERT for Flaky Test Classification

CodeBERT was pre-trained with a huge number of parameters, enabling it to recognize the source code structure. As a result, if CodeBERT were to be trained from scratch on our dataset, it would result into over-fitting. To avoid that, CodeBERT, similar to other LMs [43], needs to be fine-tuned using data representative of the problem at hand. To do this, we employed CodeBERT as pre-trained and use its outputs, on our dataset, to train a Feedforward Neural Network (FNN) to perform binary classification of test cases as flaky or non-flaky, as shown in Figure 2.2.

The output of CodeBERT, i.e., the aggregated vector representation of the `[CLS]` token, is then fed as input to a trained FNN to classify test cases as flaky or not. The FNN contains an *input* layer of 768 neurons, a *hidden* layer of 512 neurons, and an *output* layer with two neurons. We used ReLU [3] as an activation function, which helps to speed up training

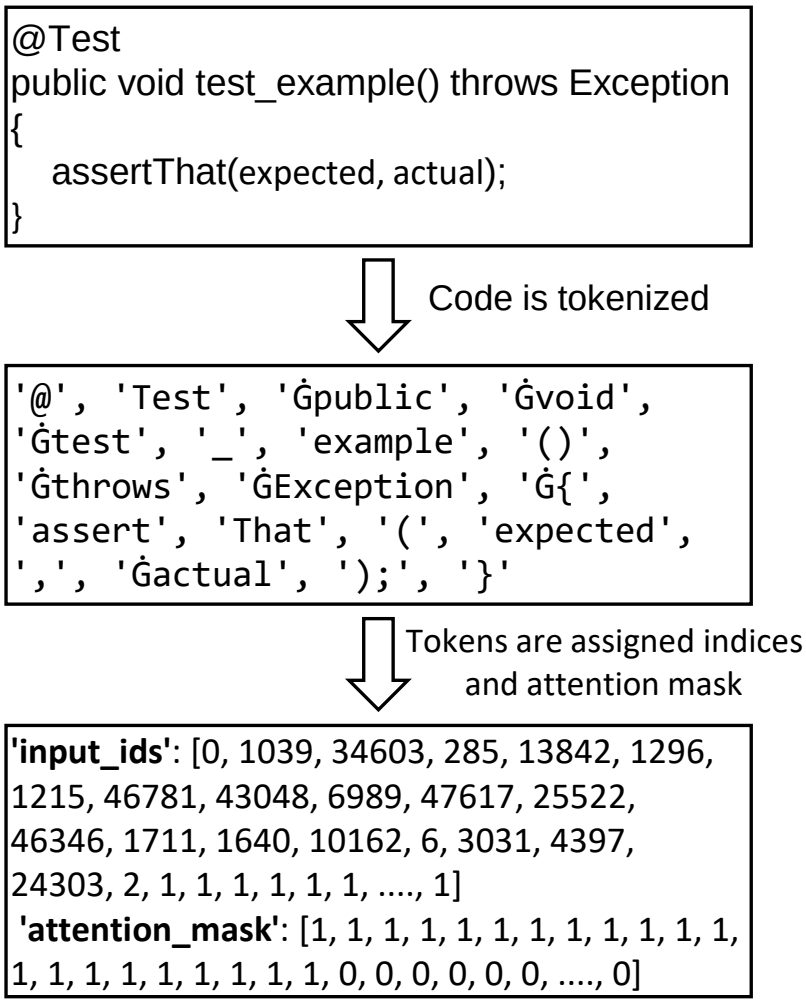



Figure 2.1: The process of converting the source code of a test case into a sequence of tokens, where each token is assigned an input index (id) and attention mask. Dots '....' are used to save space, since the actual length is 512. The input id of each token refers to a 768-dimensional vector representation.

by transforming the data within layers and output the *input* directly if it is positive or zero otherwise. Then, we added a *dropout* layer [105] to eliminate some neurons randomly from the network, by resetting their weights to zero during the training phase to prevent model over-fitting [22]. We used the *Softmax* function to compute the probability of a test case to be *Flaky* or *Non-Flaky*. We used a learning rate of $10^{-5}$ using the AdamW optimizer [125] and employed a batch size of two due to computational limitations. Using this configuration, we further trained CodeBERT on our training and validation datasets, which enabled the selection of improved parameter values for weights and biases through back propagation. We then evaluated the model, with the obtained weights, using a test dataset.

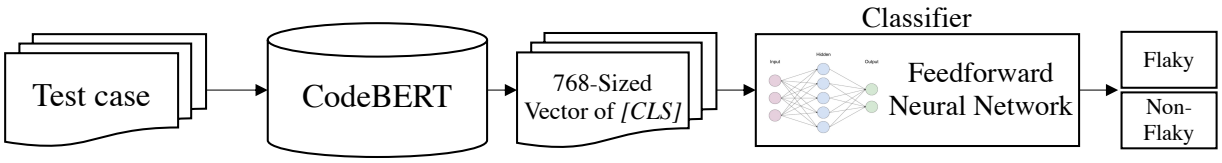



Figure 2.2: Fine-tuning CodeBERT for classifying test cases as flaky or not

### 2.3.2 Identifying Test Smells

As indicated above, the 512 token length limit induced by CodeBERT truncates longer test code, which leads to losing potentially relevant information about test flakiness. Therefore, we pre-possessed the source code of test cases to reduce their token length by only retaining information believed to be more relevant to test flakiness. To this end, for test cases exceeding the token length limit, we retained only code statements that match at least one of the eight test smells that were used by FlakeFlagger [6] as predictors for flaky test cases. We also retained the method declaration and the associated Javadoc, since the signature and natural language description, if any, of the test case, might contain key terms or phrases that are likely associated with test flakiness, e.g., *"...failures...unnecessary..."* or *"thread-safe"*.

There exist several open source tools available for detecting test smells [5]. However, these tools, e.g., *tsDetect* [91] and *JNose Test* [112], either rely on production code for detecting test smells or do not detect all test smells that are potentially relevant to test flakiness [6]. While Alshammari et al. [6] detects all the eight test smells shown in Table 2.1, their technique does so by running test cases and requiring access to the production code for smell detection. Though we were inspired by the heuristics used by Alshammari et al. to detect test smells, given that our approach aims to be black-box, we developed an entirely different technique that detects test smells statically, relying exclusively on test code without requiring to run test cases. Flakify detects all targeted test smells and can be easily extended to detect additional test smells. We used an Abstract Syntax Tree (AST) [78] parser, provided by the Eclipse JDT library,[2] to statically traverse any given test code and retain statements that match any of the targeted test smells.Using this library, each Java file in a test suite is parsed and converted into AST nodes representing different code elements, e.g., method declaration or invocation. Then, an AST visitor is used to traverse those AST nodes. We extended the AST visitor to check the AST nodes related to method declarations and apply heuristics (described below) to detect and retain code statements that match at least one test smell. Such statements are extracted as part of the pre-processed code.

Figure 2.3 gives an example of a Java test method, `test_example`, and how it is pre-processed. As we can see, `test_example` has seven different statements, four of them having test smells. In particular, `test_example` contains the following test smells: *Fire and Forget* (line 5 – launching a thread), *Conditional Test* (line 7 – `if` condition), and *Assertion Roulette* (lines 8 and 10 – multiple assertions). As a result, our technique retains

[2] `https://www.eclipse.org/jdt`

only these four statements, which in turn leads to reducing the token length from 62 to 43 (31% reduction rate). We expect our test code pre-processing to help improve the classification performance, since it mitigates the random truncation of code statements induced by CodeBERT.

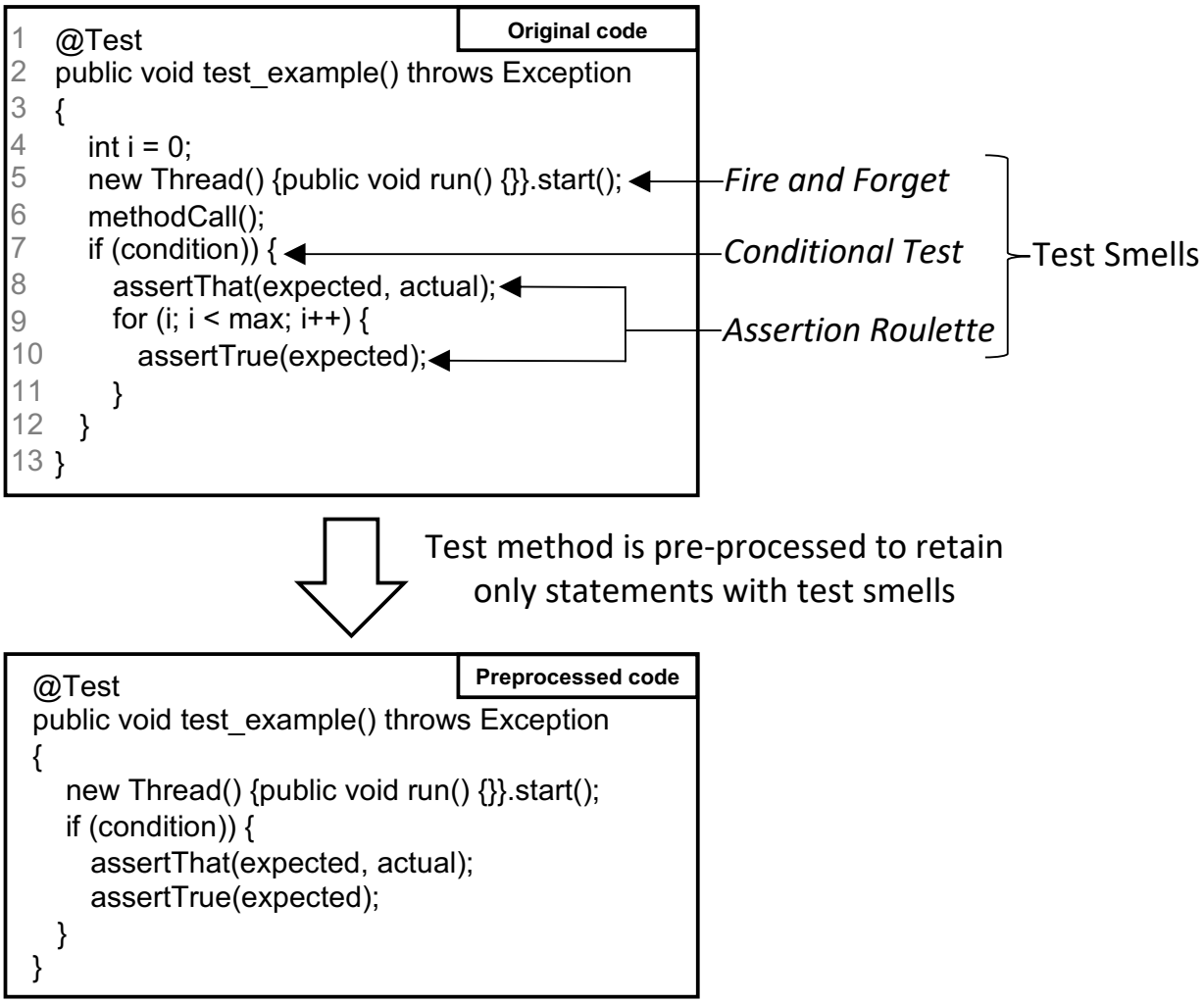



Figure 2.3: Example of pre-processing the source code of a test case, which leads to reducing the number of tokens from 62 down to 43

### Heuristics for Detecting Test Smells

To detect test smells in test code, we followed the same detection heuristics as those used by Alshammari et al. [6]. However, different from this work, which extracts test smell information dynamically from the test and production code (code coverage), we detected test smells statically by analyzing the test code only. To this end, we used an Abstract Syntax Tree (AST) [78] parser, provided by the Eclipse JDT library,[3] to traverse any given test code and retain statements that match, according to our heuristics, any of the

[3] `https://www.eclipse.org/jdt`

targeted test smells. Using this library, each Java test file in the test suite is parsed and converted into AST nodes representing different code elements, e.g., method declaration or invocation. While parsing Java test files, not all types are necessarily resolved due to missing production code. We describe below the heuristics used to identify each of the eight test smells presented in Table 2.1. For each test case, i.e., test method, we analyzed each statement to check whether it matches one of the targeted test smells. If so, we retain that statement as part of the pre-processed test code and otherwise exclude that statement. For some test smells, we added flags, i.e., a Java line comments appended to the end of each statement matching the test smell, to help our fine-tuned model learn about the association of these statements with test flakiness. The test smells used in this work were detected as described below.

- **Indirect Testing:** We check whether a statement invokes a method that belongs to a class other than the test class or the production class under test. Since our approach is black-box, i.e., no access to production code, the production class name is extracted from the test class name by removing the word '`Test`'. This is a commonly used coding convention, but our approach can easily be adapted to other coding conventions in practice [83]. Any statement that is found to invoke such methods is retained and the '`//IT`' flag is added.

- **Eager Testing:** We check whether a test case invokes more than one method belonging to the production class under test as it can introduce confusion to what exactly a test method is testing [5]. If this is the case, we retain the statements invoking these methods, adding the '`//ET`' flag.

- **Test Run War:** We check whether a statement accesses `static` variables that are not declared as *final*, as the value of such variables could be changed by other test cases in different test executions, especially when a test case is order-dependent, which can then cause resource interference during test case execution [6]. Any statement that is found to use one of these variables is retained, adding the '`//RW`' flag.

- **Conditional Logic:** We check whether a statement contains an `if` condition. If so, we retain `if` statements, including their logical expressions. The presence of conditional statements makes test case behavior dependent on their logical expressions, thus making them unpredictable [5]. For the statements inside the `then` and `else` blocks, we only retain those that match one of the eight test smells.

- **Fire and Forget:** We check whether a statement invokes a method that launches a thread by checking if the invoked method belongs to the `java.lang.Thread` class, `java.lang.Runnable` interface, or `java.util.concurrent` package. Thread-related statements make test cases prone to synchronization issues during their execution [94]. If this test smell is present, we retain that statement.

- **Mystery Guest:** We check whether a statement invokes a method that accesses external resources, such as the file system (via `java.io.File`), database system (via `java.sql`, `javax.sql`, or `javax.persistence`), or network (via `java.net` or `javax.net`). Such external resources can introduce stability and performance issues

during test case execution [94]. Any statement that is found to use methods that belong to one of these classes or packages is retained.

- **Assertion Roulette:** We check whether a statement performs one of the following assertion mechanisms, including `assertArrayEquals`, `assertEquals`, `assertFalse`, `assertNotNull`, `assertNotSame`, `assertNull`, `assertSame`, `assertThat`, `assertTrue`, and `fail`. If so, the statement is retained. Multiple assert statements in a test method makes it difficult to identify the cause of the failure if just one of the asserts fails [13].
- **Resource Optimism:** We check whether a statement accesses the file system (`java.io.File`) without checking if the path (for either a file or directory) exists. Doing so makes optimistic assumptions about the availability of resources, thus causing non-deterministic behavior of the test case [91]. We check the test initialization method (usually named as `setUp` or containing the `@Before` annotation) for any path checking method, including `getPath()`, `getAbsolutePath()`, or `getCanonicalPath()`. If no such checking is present, the statement is retained, adding the '`//RO`' flag.

## 2.4 Validation

This section reports on the experiments we conducted to evaluate how accurate is Flakify in predicting flaky test cases and how it compares to FlakeFlagger as a baseline. We discuss the research questions we address, the datasets used, and the experiment design. Then, we present the results for each research question and discuss their practical implications.

### 2.4.1 Research Questions

- **RQ1: How accurately can Flakify predict flaky test cases?**

  The performance of ML-based flaky test predictors can be influenced by the data used for training and the underlying modeling methodology. In this RQ, we evaluate Flakify on two distinct datasets, which differ in terms of numbers of projects, ratios of flaky and non-flaky test cases, and the way flaky test cases were detected. In addition, predicting flaky test cases can be influenced by project-specific information used during model training, which is not available for new projects. Therefore, we evaluate Flakify using two different procedures: 10-fold cross-validation and per-project validation. The former mixes test cases from all projects together to perform model training and testing, whereas the later tests the model on every project such that no information from that project was used as part of model training.

- **RQ2: How does Flakify compare to the state-of-the-art predictors for flaky test cases?**

  Many solutions have been proposed to predict flaky test cases. In this RQ, we compare the performance of our best performing model of Flakify (with test case

pre-processing) to two versions (white-box and black-box) of FlakeFlagger, the best flaky test case predictor to date.

***RQ2.1: How accurate is Flakify for flaky test case prediction compared to the best white-box ML-based solution?*** White-box prediction of flaky test cases requires access to production code, which is not (easily) accessible by software test engineers in many contexts. We assess whether Flakify achieves results that are at least comparable to the best white-box flaky test case predictor. Specifically, we compare the accuracy of the best performing model of Flakify with FlakeFlagger [6], the best white-box solution currently available, on the dataset used by FlakeFlagger. Our motivation is to determine whether black-box solutions, based on CodeBERT, can compete with the state-of-the-art, white-box ones. We compare the results of Flakify and FlakeFlagger on the dataset on which FlakeFlagger was evaluated, hereafter referred to as the FlakeFlagger dataset. We also performed a per-project validation of Flakify compared against FlakeFlagger to assess their relative capability to predict test cases in new projects.

***RQ2.2: How accurate is Flakify for black-box flaky test case prediction compared to the best ML-based solution?*** Existing black-box flaky test case prediction solutions rely on a limited set of features that are sometimes project-specific or applicable only to a certain programming language, e.g., Java [93], since they were trained on features capturing the keywords of that language. Besides not being generic, the accuracy of these solutions has shown to be very low compared to white-box solutions [6]. Therefore, we compare the accuracy of Flakify with a black-box version of FlakeFlagger, by excluding the features related to production code, such as code coverage features (see Table 2.2).

- **RQ3: How does test case pre-processing improve Flakify?** The token length limitation of CodeBERT may lead to unintentionally removing relevant information about flaky test cases, which could then impact prediction accuracy. We assess whether the accuracy of Flakify is improved when training the model using pre-processed test cases containing only code statements related to test smells, as opposed to the entire test case code. We fully realize that we may be missing test smells or unintentionally removing relevant statements. But our motivation is to assess the benefits, if any, of our approach to reduce the number of tokens used as input to CodeBERT. We performed this analysis on both the FlakeFlagger and the IDoFT datasets.

### 2.4.2 Datasets Collection and Processing

To evaluate Flakify, we used two publicly available datasets for flaky test cases. The first dataset is the FlakeFlagger dataset [6]. The second dataset is the International Dataset of

| Category | Feature | Description |
|---|---|---|
| Black-Box | Presence of Test Smells | See Table 2.1 |
| | Test Lines of Code | Number of lines of code in the body of the test method |
| | Number of Assertions | Number of assertions checked by the test |
| | Execution Time | Running time for the test execution |
| | Libraries | Number of external libraries used by the test |
| White-Box | Source Covered Classes | Number of production classes covered by each test |
| | Source Covered Lines | Number of lines covered by the test, counting only production code |
| | Covered Lines | Number of lines of code covered by the test |
| | Covered Lines Churn | Churn of covered lines in past 5, 10, 25, 50, 75, 100, 500, and 10,000 commits |

Table 2.2: FlakeFlagger Features

Flaky Tests (IDoFT)[4], which comprises many datasets for flaky test cases used by previous studies on flaky test case prediction [53–56, 103, 115].

**FlakeFlagger dataset:** It is provided by Alshammari et al. [6], containing flakiness information about 22,236 test cases collected from 23 GitHub projects. These projects have different test suite sizes, ranging from 55 to 6,267 (with a median of 430) test cases per project. All projects in the FlakeFlagger dataset are written in Java and use Maven as a build system, and each test case is a Java test method. The dataset contains the source code of each test case and the corresponding features that were computed to train FlakeFlagger. Also, test cases in the dataset were assigned labels indicating whether they are *Flaky* or *Non-Flaky*, which were determined by executing each test case 10,000 times.

When we analyzed the dataset, we identified 453 test cases with missing source code when intersecting test cases in a provided CSV file (called *processed_data*[5]) with those in a provided folder (called *original_tests*[6]) containing their source code. In addition, we identified 122 test cases, in the *original_tests* folder, with empty source code, which we found out were not written in Java.[7] Therefore, we excluded these test cases from our dataset, since they do not add any valuable information regarding our flakiness prediction evaluation. Nine of these test cases were labeled as flaky, three with missing source code and six with empty method body. After excluding test cases with missing and empty code, we obtained 21,661 test cases for our experiments. We compared Flakify and FlakeFlagger using this updated dataset. To pre-process the source code of the test cases (see Section 2.3.2), we cloned the GitHub repository of each project and extracted the Java classes defining the methods of test cases.

There are 802 test cases in the dataset that are labeled as *Flaky* (with a median of 19 flaky test cases per project), whereas 20,859 test cases are *Non-Flaky*. About 4% of all test cases exceed the 512 limit of CodeBERT when converted into tokens, including 14% of the flaky test cases.

**IDoFT dataset:** This dataset contains 3,742 *Flaky* test cases from 314 different Java

[4] `https://mir.cs.illinois.edu/flakytests`

[5] `https://github.com/AlshammariA/FlakeFlagger/blob/main/flakiness-predicter/result/processed_data.csv`

[6] `https://github.com/AlshammariA/FlakeFlagger/tree/main/flakiness-predicter/input_data/original_tests`

[7] `https://github.com/AlshammariA/FlakeFlagger/pull/4`

projects, and collected using different ways, i.e., different runtime environments with different numbers of runs to detect test flakiness. However, we were unable to obtain the test code of 474 test cases (from 2 projects) due to missing GitHub repositories or commits, leaving us with 3,268 *Flaky* test cases from 312 projects. Given that the IDoFT dataset contains no test cases categorized as *Non-Flaky*, we used the fixed versions of 1,263 flaky test cases, from 174 projects, to obtain non-flaky test cases, as recommended by the IDoFT maintainers[8]. To do so, we relied on the provided links to pull requests[9] used for fixing flaky test cases to collect the corresponding code changes. However, of the 1,263 fixed flaky test cases, we found only 594 flaky test cases, from 126 projects, in which the test case code is changed to fix test flakiness. Based on our analysis, the other flaky test cases were fixed in other ways, such as changing the order of test case execution, test configuration, or production code. Such flaky tests are out of the scope of this research, since we consider only test cases whose test code was fixed, e.g., causes of flakiness related to test smells or other test characteristics. As a result, we added the 594 *Non-Flaky* (fixed) tests to the 3,268 *Flaky* test cases to end up with an updated dataset of 3,862 test cases. Limitations, regarding the causes of flakiness we could not detect, are discussed in Section 2.5. About 13% of all test cases exceed the 512 limit of CodeBERT when converted into tokens.

We made the updated datasets of FlakeFlagger and IDoFT, including their pre-processed test cases, publicly available in our replication package [25].

### 2.4.3 Experiment Design

#### Baseline

We used the FlakeFlagger approach as a baseline against which we compare the results achieved by Flakify. To this end, we reran the experiments conducted by Alshammari et al. [6] to reproduce the prediction results of FlakeFlagger using their provided replication package.[10] FlakeFlagger was trained and tested using a combination of white-box and black-box features listed in Table 2.2. These features were selected based on their Information Gain (IG), i.e., only features having an IG $\geq 0.01$ were selected for training. Besides reproducing the original results of FlakeFlagger, we also reran the experiments using black-box features only, which was done by excluding all features that required access to production code. Comparing Flakify with FlakeFlagger is performed on the FlakeFlagger dataset only, as running FlakeFlagger on the IDoFT dataset requires extracting features, both dynamic and static, needed to train FlakeFlagger. To do so, we must access the project's production code and then successfully execute thousands of test cases across hundreds of project versions.

[8] `https://github.com/TestingResearchIllinois/IDoFT/issues/566`
[9] `https://mir.cs.illinois.edu/flakytests/fixed.html`
[10] `https://github.com/AlshammariA/FlakeFlagger`

### Training and Testing Prediction Models

Training and testing Flakify were conducted using two different procedures, performed independently on the two datasets describe above, as follows.

**1$^{st}$ Procedure (cross-validation):** In this procedure, we evaluated Flakify similarly to how FlakeFlagger was originally assessed. Specifically, we used a 10-fold stratified cross-validation to ensure our model is trained and tested in a valid and unbiased way. For that, we allocated 90% of the test cases for training and 10% for testing our model in each fold. However, different from FlakeFlagger, we employed 20% of the training dataset as a validation dataset, which is required for fine-tuning CodeBERT. Using the validation dataset, we calculated the training and validation loss, which helped obtain optimal weights and stop the training early enough to avoid overfitting.

Given that both of the datasets we used are highly imbalanced—*Flaky* test cases represent only 3.7% of all test cases in the FlakeFlagger dataset and *Non-Flaky* test cases represent only 15% of the IDoFT dataset—we balanced *Flaky* and *Non-Flaky* test cases in the training and validation datasets of FlakeFlagger and IDoFT. Different from FlakeFlagger, which used the synthetic minority oversampling technique (SMOTE) [15], we used random oversampling [11], which adds random copies of the minority class to the dataset. We were unable to use SMOTE, since it requires vector-based features, whereas our model takes the source code of test cases (text) as input [29, 82], as opposed to pre-defined features like FlakeFlagger. Similar to FlakeFlagger, we also performed our experiments using undersampling but this led to lower accuracy. We did not balance the testing dataset to ensure that our model is only tested on the actual set of test cases. This prevents overestimating the accuracy of the model and reflects real-world scenarios where flaky test cases are rarer than non-flaky test cases [6].

**2$^{nd}$ Procedure (per-project validation):** In this procedure, we evaluated Flakify in a way that yields more realistic results when we predict test cases on a new project, thus evaluating the generalizability of Flakify across projects. To do this, we performed a per-project validation of Flakify on both datasets. In particular, for every project in each dataset, we trained Flakify on the other projects and tested it on that project. This allowed us to evaluate how accurate Flakify is in predicting flaky test cases in one project without including any data from that project during training. We also performed this analysis for FlakeFlagger, on the FlakeFlagger dataset, for the sake of comparison.

### Evaluation Metrics

To evaluate the performance of our approach, we used standard evaluation metrics for ML classifiers, including *Precision* (the ability of a classification model to precisely predict flaky test cases), *Recall* (the ability of a model to predict all flaky test cases), and the *F1-Score* (the harmonic mean of precision and recall) [33]. For the per-project validation of Flakify, we computed the overall precision, recall, and F1-score using the prediction results of all projects in the FlakeFlagger and IDoFT datasets. We also computed these metrics individually for those projects that have both *Flaky* and *Non-Flaky* test cases, specifically

23 FlakeFlagger projects and 126 IDoFT projects, along with descriptive statistics, such as mean, median, min, max, 25% and 75% quantiles. We used Fisher's exact test [97] to assess how significant is the difference in proportions of correctly classified test cases between two independent experiments. Note that precision, recall, and F1-score are computed based on such proportions.

In practice, test cases classified as *Flaky* must be addressed by re-running them multiple times or by fixing the root causes of flakiness [70, 71, 130]. Precisely predicting flakiness is therefore important as otherwise time and resources are wasted on re-running and attempting to debug many test cases that are believed to be flaky but are not [66, 85]. According to our industry partner, Huawei Canada, and a Google technical report [71], each flaky test case has to be investigated and re-run by developers. Hence, when we multiply the number of predicted flaky test cases, we proportionally increase the resources associated with re-running and investigating such flaky test cases. Therefore, we assume that the wasted cost of unnecessarily re-running and debugging test cases is inversely proportional to precision:

$$\textit{Test Debugging Cost} \propto 1 - \textit{Precision} \tag{2.1}$$

On the other hand, it is also important not to miss too many flaky test cases as otherwise time is bound to be wasted on futile attempts to find and fix non-existent bugs in the production code. Thus, we assume that the wasted cost of unnecessarily finding and fixing non-existent bugs in the production code is inversely proportional to recall:

$$\textit{Code Debugging Cost} \propto 1 - \textit{Recall} \tag{2.2}$$

We acknowledge that the above metrics are surrogate measures for cost and that there are significant differences between individual flaky tests; however, they are reasonable and useful approximations on large test suites for the purpose of comparing classification techniques. We used FlakeFlagger as baseline to compute the reduction rate of test and code debugging costs, by dividing the difference in cost between Flakify and FlakeFlagger by the cost of FlakeFlagger.

### 2.4.4 Results

#### RQ1 results

Table 2.3 shows the prediction results (in terms of precision, recall, and F1-score) of Flakify using both the full and pre-processed test code from the FlakeFlagger and IDoFT datasets, based on cross-validation. Overall, Flakify achieved promising prediction results using both datasets, with a precision of 70%, a recall of 90%, and an F1-score of 79% on the FlakeFlagger dataset, and a precision of 99%, a recall of 96%, and an F1-score of 98% on the IDoFT dataset. The higher results achieved by Flakify on the IDoFT dataset over those achieved on the FlakeFlagger dataset is probably due to the fact that the IDoFT

dataset contains many more flaky test cases than FlakeFlagger, which helped during model training. Moreover, the non-flaky test cases in the IDoFT dataset were labeled based on developer's fixes addressing the causes of flakiness in the test code, unlike the non-flaky test cases in the FlakeFlagger dataset whose labels were based on 10,000 runs performed by Alshammari et al. [6], which may not have been enough to fully expose test flakiness. This also helped during model training of Flakify.

Table 2.4 reports the per-project prediction results of Flakify on the FlakeFlagger dataset. Overall, as expected, Flakify achieved slightly lower precision (72%), recall (85%), and F1-score (73%) than the cross-validation results on the FlakeFlagger dataset. Similarly, Flakify achieved slightly worse precision (91%), recall (88%), and F1-score (89%) on the IDoFT dataset. Table 2.5 shows descriptive statistics for the per-project prediction results of Flakify for individual projects of the FlakeFlagger dataset (due to space limitations, we provide individual per-project prediction results of Flakify on the IDoFT dataset in our replication package [25]). Our analysis of individual per-project prediction results revealed a high performance of Flakify on the majority of projects. This result suggests that Flakify helps build models that are generalizable across projects, thus making it applicable to new projects where no historical information about test flakiness exists. In short, Flakify is capable to learn about test flakiness through data collected from other projects to predict flaky test cases in new projects.

| **Approach** | **Dataset** | **Model** | **Precision** | **Recall** | **F1-Score** |
|---|---|---|---|---|---|
| Flakify | FlakeFlagger dataset | Full code | 65% | 85% | 74% |
| | | Pre-processed code | **70%** | **90%** | **79%** |
| | IDoFT dataset | Full code | 98% | 95% | 92% |
| | | Pre-processed code | **99%** | **96%** | **98%** |
| FlakeFlagger | FlakeFlagger dataset | White-box version | 60% | 72% | 65% |
| | | Black-box version | 21% | 52% | 30% |

Table 2.3: Results of Flakify (using full code and pre-processed) compared to FlakeFlagger (white-box and black-box versions)

### RQ2 results

Table 2.3 presents the prediction results of Flakify, using both full code and pre-processed test code, and FlakeFlagger, using both white-box and black-box versions, for the FlakeFlagger dataset.

***RQ2.1 results.*** For FlakeFlagger, we obtained results close to those reported in the original study, with a slight decrease in F1-score (1%), which is likely due to removing test cases with missing test code. Flakify achieved much better results with a precision of

| Dataset | Metric | Min | 25% | Mean | Median | 75% | Max |
|---|---|---|---|---|---|---|---|
| FlakeFlagger | Precision | 6% | 58% | **72%** | 79% | 91% | 100% |
| | Recall | 1% | 87% | **85%** | 95% | 100% | 100% |
| | F1-Score | 2% | 63% | **73%** | 83% | 94% | 100% |
| IDoFT | Precision | 66% | 100% | **91%** | 100% | 100% | 100% |
| | Recall | 14% | 94% | **88%** | 100% | 100% | 100% |
| | F1-Score | 25% | 95% | **89%** | 100% | 100% | 100% |

Table 2.4: Summary of the per-project prediction results of Flakify on the FlakeFlagger and IDoFT datasets

70% (+10 pp), a recall of 90% (+18 pp), and an F1-score of 79% (+14 pp). These results clearly show that Flakify, though being black-box and relying exclusively on test code, significantly surpasses FlakeFlagger in accurately predicting flaky test cases. Statistically, the proportion of correctly predicted test cases using Flakify is significantly higher than that obtained with FlakeFlagger (Fisher-exact p-value $< 0.0001$).

The number of true positives obtained by FlakeFlagger was 574, whereas Flakify increased that number to 721. This indicates that Flakify can potentially reduce the test debugging cost by 10 pp, as defined above, when compared to FlakeFlagger (a reduction rate of 25%). Similarly, Flakify reduces the number of false negatives to 81 from 227 with FlakeFlagger, thus decreasing the code debugging cost by 18 pp, as defined above (a reduction rate of 64%).

Table 2.5 shows the comparison of per-project prediction results between Flakify and FlakeFlagger. Overall, Flakify achieves a high accuracy, with a precision of 72% (+57 pp)), a recall of 85% (+71 pp)), and an F1-score of 73% (+66 pp)), which, once again, significantly outperforms FlakeFlagger. Looking at the individual prediction results of the projects, we observe that the accuracy of Flakify is largely consistent across projects, with a few exceptions, whereas FlakeFlagger performed poorly on the majority of projects. Further, Flakify performs better than FlakeFlagger for almost all projects except two: `incubator-dubbo` and `spring-boot` where both techniques fare poorly.

To understand the reasons behind such degraded performance for these two projects, we performed a hierarchical clustering of the 23 projects. We used different metrics that capture the characteristics of each project, such as the number of test cases, number of flaky test cases, and frequency of test smells in each project. However, our clustering results were inconclusive, thus revealing no significant differences between the two projects and the other projects. As reported by Alshammari et al. [6], each project can have distinct characteristics, e.g., environmental setup and testing paradigm, that make it difficult to develop a general-purpose flaky test case predictor. For example, the `spring-boot` project has the highest number of flaky test cases among all projects, representing 20% of all flaky

| Project | Precision | | Recall | | F1-Score | |
|---|---|---|---|---|---|---|
| | **Flakify** | **FlakeFlagger** | **Flakify** | **FlakeFlagger** | **Flakify** | **FlakeFlagger** |
| achilles | 100% | 0% | 100% | 0% | 100% | 0% |
| activiti | 80% | 2% | 90% | 94% | 85% | 4% |
| alluxio | 99% | 100% | 100% | 13% | 99% | 24% |
| ambari | 75% | 39% | 95% | 61% | 84% | 47% |
| assertj-core | 25% | 0% | 100% | 0% | 40% | 0% |
| commons-exec | 25% | 0% | 100% | 0% | 40% | 0% |
| elastic-job-lite | 50% | 0% | 100% | 0% | 60% | 0% |
| handlebars.java | 30% | 0% | 100% | 0% | 50% | 0% |
| hbase | 79% | 72% | 98% | 33% | 88% | 45% |
| hector | 100% | 0% | 93% | 0% | 96% | 0% |
| http-request | 88% | 0% | 88% | 0% | 88% | 0% |
| httpcore | 74% | 7% | 90% | 4% | 81% | 5% |
| incubator-dubbo | 6% | 7% | 16% | 32% | 9% | 12% |
| java-websocket | 95% | 0% | 95% | 0% | 95% | 0% |
| logback | 85% | 0% | 81% | 0% | 83% | 0% |
| ninja | 100% | 0% | 100% | 0% | 100% | 0% |
| okhttp | 78% | 100% | 85% | 2% | 81% | 4% |
| orbit | 88% | 0% | 100% | 0% | 93% | 0% |
| spring-boot | 40% | 9% | 1% | 3% | 2% | 4% |
| undertow | 75% | 7% | 85% | 43% | 79% | 12% |
| wildfly | 65% | 6% | 91% | 26% | 76% | 10% |
| wro4j | 88% | 1% | 100% | 19% | 94% | 3% |
| zxing | 100% | 0% | 50% | 0% | 66% | 0% |
| Overall | **72%** | 15% | **85%** | 14% | **73%** | 7% |

Table 2.5: Results of the per-project prediction for Flakify and FlakeFlagger on the FlakeFlagger dataset

test cases in the dataset. This, in turn, can influence model training when the model was tested for `spring-boot`. In addition, the variation in prediction results can be a result of a possible mislabeling of test cases as *Flaky* and *Non-Flaky* in some projects, since some test cases may still exhibit flakiness behavior if executed more than 10,000 executions, for example. Finally, test flakiness can also occur due to the use of network APIs or dependency conflicts [86], which were not taken into account when predicting flaky test cases.

***RQ2.2 results.*** As shown in Table 2.3, we observe a considerable decline in the accuracy for the black-box version of FlakeFlagger when compared to its original, white-box version, i.e., 39 pp less precise with a 54 pp decrease in F1-score. Specifically, black-box FlakeFlagger correctly predicted a significantly lower proportion of test cases than both Flakify and the original, white-box version of FlakeFlagger (Fisher-exact p-values $< 0.0001$). As a possible explanation, based on the results of FlakeFlagger regarding the importance of features in predicting flaky test cases [6], the majority of features having high IG values were based on source code coverage. Hence, removing those features, to make FlakeFlagger black-box, is expected to significantly decrease its prediction power. The difference in accuracy between Flakify and the black-box version of FlakeFlagger is rather striking, with a large improvement of +49% in F1-score (Fisher-exact p-value $< 0.0001$). FlakeFlagger is therefore not a viable black-box option to predict flaky test cases.

**RQ3 results**

With no code pre-processing, 898 (4%) of the test cases of the FlakeFlagger dataset and 505 (13%) of the test cases of the IDoFT dataset were truncated by CodeBERT to generate tokens of size 512. Such arbitrary code truncation is likely to affect how accurately Flakify can predict flaky test cases. Pre-processing test cases (see Section 2.3.2) led to reducing the number of test cases being truncated to only 40 (from 898) in the FlakeFlagger dataset and 87 (from 505) in the IDoFT dataset, a large difference. As a result, we observe in Table 2.3 that, with pre-processed test cases, Flakify predicted flaky test cases with 5 pp higher F1-score on the FlakeFlagger dataset and 6 pp higher F1-score on the IDoFT dataset. This corresponds to a significantly higher proportion of correctly predicted test cases (Fisher-exact p-value = 0.0008) for the FlakeFlagger dataset. In practice, the impact of pre-processing is expected to vary depending on the token length distribution of test cases. This result suggests that retaining statements related to test smells in the test code contributed to making Flakify more accurate, which also confirms the association of test smells with flaky test cases reported by prior research [13].

### 2.4.5 Discussion

**More accurate predictions with easily accessible information.** Our results showed that our black-box prediction of flaky test cases performs significantly better than a white-box, state-of-the-art approach. This not only enables test engineers to predict flaky test cases without rerunning test cases, but also without accessing the production code of

the system under test, a significant practical advantage in many contexts. The highest accuracy of our Flakify was achieved by only retaining relevant code statements matching eight test smells. Yet, there is still room for improvement in terms of accuracy, which could be achieved by retaining more relevant statements based on additional test smells. For example, retaining code statements related to other common flakiness causes [85], such as concurrency and randomness, could further improve flaky test case predictions. However, the more code statements we retain, the more tokens to be considered by CodeBERT, which might lead to many test cases exceeding their token length limit, thus truncating other useful information. Hence, retaining additional code statements is a trade-off and should carefully be performed in balance with the resulting token length of test cases. Moreover, building a white-box flaky test predictor, by considering both production and test code, is not always technically feasible, since the production code is not always available to test engineers and, when possible, code coverage can be expensive and not scalable on large systems, especially in a continuous integration context. Considering the production code also makes it impractical to build LM-based predictors for flaky test cases, given the token length limitation of LMs in general, and CodeBERT in particular. Nevertheless, future research should assess the practicability of white-box, model-based flaky test prediction, and should investigate further code pre-processing methods to make the use of LMs more applicable in practice.

**Practical implications of imperfect prediction results.** Though Flakify surpassed the best state-of-the-art solution in predicting flaky test cases, both in terms of precision and recall, a precision of 70% is still not satisfactory, since misclassifying non-flaky test cases as flaky leads to additional, unnecessary cost, e.g., attempting to fix the test cases incorrectly predicted as flaky. Also, with a recall of 90%, we miss 10% of flaky test cases, leading to wasted debugging cost. If we assume that precision should be prioritized over recall, we can increase the former by restricting flaky test case predictions to those test cases with highest prediction confidence, at the expense of a lower recall. For example, this can be achieved by adjusting the classification threshold for flaky test cases to 0.60 or 0.70, instead of the default threshold of 0.50. Nevertheless, given that the predicted probabilities generated by the neural network in Flakify are over confident due to the use of the *Softmax* function in the last layer [65], i.e., probabilities are either close to 0.0 or 1.0, we were unable to perform such analysis. Therefore, future research should employ techniques for calibrating the predicted probabilities [35] and enable threshold adjustments when classifying flaky test cases.

## 2.5 Threats to Validity

This section discusses the potential threats to the validity of our reported results.

### 2.5.1 Construct Validity

Construct threats to validity are concerned with the degree to which our analyses measure what we claim to analyze. In our study, to pre-process test cases, we used heuristics to

retain code statements that match at least one of the eight test smells shown in Table 2.1. However, our heuristics might have missed some code statements having test smells and this could have led to suboptimal results when applying our approach. To mitigate this issue, though our approach to identify test smells is entirely different, we relied on the same heuristics as those used by Alshammari et al. [6]. These heuristics assume commonly used coding conventions that might not be followed in all test suites. For example, we assumed that the test class name contains the production class name with the word '*Test*'. However, such heuristics can easily be adapted to other coding conventions in practice. We also manually checked a random sample of test cases to verify that pre-processed code contains, as expected, only test smells-related code statements and does not dismiss any of them. We have made the tool we developed to detect test smells publicly available in our replication package [25].

### 2.5.2 Internal Validity

Internal threats to validity are concerned with the ability to draw conclusions from our experimental results. In our study, we used CodeBERT to perform a binary classification of test cases as *Flaky* or *Non-Flaky*. However, due to the token length limit of CodeBERT, the source code of some test cases was truncated, possibly leading to discarding relevant information about test flakiness. To mitigate this issue, we pre-processed the source code of test cases to retain only code statements related to test smells. Doing so did not only reduce the token length of test cases, but also improved the prediction power of our approach. However, our pre-processing may not be perfect or complete as it can lead to losing other relevant information. Future research should investigate whether retaining additionally relevant information to flaky test cases leads to improving prediction results, e.g., statements related to common flakiness causes, such as synchronous or platform-dependent operations.

Moreover, our prediction results were compared with those of FlakeFlagger. But FlakeFlagger used white-box features, whereas our approach is black-box and the comparison may not be entirely meaningful. To mitigate this issue, we also compared our results with a black-box version of FlakeFlagger in which we removed any features requiring access to production code. In both cases, our approach obtained significantly higher prediction results than FlakeFlagger. We did not compare our results with other black-box approaches, e.g., vocabulary-based [93], since they are project-specific and did not achieve good results on the FlakeFlagger dataset [6].

Finally, in our analysis, the cost of debugging the production or testing code assumes that test engineers address all test cases predicted as flaky. However, test engineers may choose to ignore a flaky test case, either by removing or skipping it, thus not introducing any cost. Yet, we believe that every flaky test case should be carefully addressed by test engineers, since ignoring test cases can lead to other kinds of costs, such as overlooked system faults.

### 2.5.3 External Validity

External threats are concerned with the ability to generalize our results. Our study is based on data collected by Alshammari et al. [6], which was obtained by rerunning test cases 10,000 times. Such data is of course not perfect as some test cases that were not found to be flaky could have been if rerun more times. To mitigate this threat, we used the same dataset for comparing Flakify with the baseline approach, FlakeFlagger. We also filtered out test cases which, to our surprise, had no source code in the dataset. Further, the FlakeFlagger and IDoFT datasets contain test cases from projects that are exclusively written in Java, which might affect the generalizability of our results. To mitigate this issue, we used CodeBERT, which was trained on six programming languages. Hence, we believe our approach would be applicable to projects written in other programming languages as well, given an appropriate tool to identify test smells.

Moreover, CodeBERT was pre-trained on production source code only, i.e., source code related to test suites was not part of pre-training, making it unable to recognize test-specific structure and vocabulary, e.g., assertions. This can potentially increase token length, since test-specific key terms are decomposed into multiple tokens instead of one. For example, CodeBERT converts `assertEquals` into three tokens: `assert`, `##equal`, and `##s`, rather than just one token. Our pre-processing of the source code of test cases helped to mitigate the issue of token length; yet, future work should aim at pre-training CodeBERT on test code in addition to production code.

Finally, the IDoFT dataset has shown that a significant number of test cases are flaky due to reasons unrelated to the test code. In situations where this is common, this is obviously a limitation of any black-box approach like Flakify relying exclusively on test code. In our evaluation, we did not consider such flaky test cases, but rather those whose causes of flakiness were in the test code, which were confirmed and manually fixed by developers, and thus considered in this research as non-flaky. This helped during model training of Flakify on this dataset, which resulted in a higher prediction accuracy than those on the FlakeFlagger dataset.

## 2.6 Related Work

Flaky test detection has been an active area of research where many techniques were proposed to detect flaky test cases [85]. Overall, these techniques can be classified into two groups: dynamic techniques, which require executing test cases to determine whether they are flaky or not, and static techniques, which rely only on the source code of test cases or the system under test. In this section, we review the flaky test detection techniques while comparing and contrasting them to our approach.

### 2.6.1 ML-based Flaky Test Case Prediction

A common approach to detect flaky test cases is to re-run test cases multiple times [85,131], which is computationally expensive. To address this issue, recent research has proposed

the use of ML techniques for predicting flaky test cases, enabling test engineers to re-run only those test cases that are predicted to be flaky, thus reducing the cost of unnecessary debugging of test cases or production code.

Alshammari et al. [6] proposed an innovative approach to predict flaky test cases using dynamically computed features capturing code coverage, execution history, and test smells. They re-ran test cases 10,000 times to identify whether a test case was flaky or not and thus establish a ground truth. Their prediction model predicted flaky test cases with an F1-score of 0.65, leaving significant room for improvement. However, some of the significant features required access to production files which, as discussed above, are not always accessible by test engineers or may not be computable in a scalable way in many practical contexts. Further, when only black-box features (see Table 2.2) were used, the F1-score decreased by 35 pp. In contrast, our approach achieved more accurate prediction results, with an F1-score of 0.79, while using test code only, thus offering a favorable black-box alternative.

In addition, Pontillo et al. [94] proposed an approach to identify the most important factors associated with flaky test cases using the iDFlakies dataset [53]. They used logistic regression to model flaky test cases using features that were statically computed using production code, e.g., code coverage, and test code, e.g., test smells. They found that code complexity (both production and test code), assertions, and test smells are associated with test flakiness.

Another approach was proposed by Pinto et al. [93] in which Java keywords were extracted from test code and employed as vocabulary features to predict test flakiness. Further, their study relied on the dataset of DeFlaker [10], in which test cases were re-run less than 100 times to establish the ground truth. Despite high accuracy results (F1-score = 0.95) on their dataset, their approach achieved much worse results (F1-score = 0.19) when using the dataset provided by Alshammari et al. [6]. In addition, their models were language- and project-specific, since most of the significant features for predicting flaky test cases were related to Java keywords, e.g., *throws*, or specific variable names, e.g., *id*.In contrast, while our approach relies exclusively on test code, it builds a generic model to predict flakiness, based on features that are neither language- nor project-dependent, and achieved much better prediction results when using the FlakeFlagger dataset used by Alshammari et al. [6].

Moreover, Haben et al. [38] and Camara et al. [14] replicated the study by Pinto et al. using other datasets containing projects written in other programming languages, e.g., Python. They found that vocabulary-based approaches are not generalizable, especially when performing inter-project flaky test case predictions, since new vocabulary is needed for any new project or programming language. Haben et al. also showed that combining the vocabulary-based features with code coverage features does not significantly improve the prediction accuracy of such an approach.

In summary, unlike the ML-based approaches above, our approach is generic, black-box, and LM-based, thus not requiring access to production code or pre-definition of features. Instead, our approach relies solely on test code to predict whether a test case is flaky or not.

### 2.6.2 Flaky Test Case Prediction using Test Smells

Camara et al. [13] proposed an approach for predicting test flakiness using test smells as prediction features. These features require access to the production code and can be extracted using tsDetect [91], a tool for detecting test smells, that was applied to the DeFlaker dataset [10]. Their study yielded a relatively high prediction accuracy (F1-score = 0.83). Alshammari et al. [6] also relied on test smells as part of their features for predicting flaky test cases. However, the information gain of test smell features tended to be much lower than code coverage features, suggesting they are less significant flaky test case predictors. In Flakify, we also relied on the test smells used by Alshammari et al. [6]. However, they were not used as features but to exclusively retain relevant test code statements for fine-tuning our CodeBERT model. Doing so improved the accuracy of Flakify, thus reducing the cost of rerunning or debugging test cases.

### 2.6.3 Flaky Test Detection at Run Time

Memon et al. [67] used a simple dynamic pattern matching approach to detect flaky test cases at Google by simply searching for certain textual patterns in test execution logs, e.g., *pass-fail-pass*, to identify whether a test case is flaky or not. The accuracy of detecting flaky test cases using this approach was 90%. Similarly, Kowalczyk et al. [49] detected flaky test cases at Apple by analyzing the behavior of test cases using two scores: *Flip rate*, which measures the rate at which a test case alternates between *pass* and *fail*, and *Entropy*, which quantifies the uncertainty of a test case. An aggregated value of these two scores was used to generate flakiness ranks for test cases, which were then used to represent test flakiness, distributed across the test cases in different services at Apple. This technique marked 44% of test failures as flaky with less than 1% loss in fault detection. The above approaches require test cases to be executed many times to determine whether they are flaky, which is often not practical for large industrial projects. Unlike these approaches, Flakify is able to predict flaky test cases without executing them, relying exclusively on test code.

Bell et al. [10] proposed DeFlaker, a tool for detecting flaky test cases using coverage information about code changes. In particular, a test case is labeled as flaky if it fails and does not cover any changed code. Out of 4,846 test failures, DeFlaker was able to label 39 pp of them as flaky, with a 95.5% recall and a false positive rate of 1.5%, outperforming the default way of detecting flaky test cases, i.e., by rerunning test cases using Maven [1]. Different from DeFlaker, Lam et al. [53] proposed iDFlakies, which detects test flakiness by re-running test cases in random orders. This framework was used to construct a dataset containing 422 flaky test cases, with almost half of them being order-dependent.

The above approaches either depend on rerunning test cases multiple times, execution history (not available for new test cases), or production code, e.g., coverage information. In contrast, Flakify does not require repeated executions of test cases or any information about the production code, including code coverage.

## 2.7 Conclusion

In this chapter, we proposed Flakify, a black-box solution for predicting flaky test cases using only the source code of test cases, as opposed to the system under test. Further, it does not require to rerun test cases multiple times and does not entail the definition of features for ML prediction.

We used CodeBERT, a pre-trained LM, and fine-tuned it to classify test cases as flaky or not based exclusively on test source code. We evaluated our work on two distinct datasets, namely the FlakeFlagger and IDoFT datasets, using two different evaluation procedures: (1) cross-validation and (2) per-project validation, i.e., prediction on new projects. In addition, we pre-processed this source code by retaining only code statements that match eight test smells, which are expected to be associated with test flakiness. This aimed at addressing a limitation of CodeBERT (and related LMs), which can only process 512 tokens per test case. We evaluated our approach in comparison with both white-box and black-box versions of FlakeFlagger, the best state-of-the-art, ML-based flaky test case predictor.

# Chapter 3

# Categorization of Flaky Tests

This chapter addresses $TO_2$, i.e. categorization of flaky tests based on their fix categories. The contents of this chapter have been published as Research Questions *RQ1* and *RQ2* in the Journal of *Transactions on Software Engineering (TSE)* [28].

## 3.1 Overview

As discussed in chapter 1, *Chapter Structure*. The rest of this section is structured as follows. Section 3.2 provides background materials on the problem of categorizing flaky tests based on the fix required. Section 3.3 discusses our approach for defining fix categories, automatically labeling the dataset, and then predicting the fix categories using code models. Section 3.4 presents the evaluation of flaky test categorization and discusses the results. Section 3.5 discusses the qualitative analysis of the model's performance. Section 3.6 reports threats to the validity of the reported results. Finally Section 3.7 concludes this chapter.

## 3.2 Background

The intended purpose of categorizing flaky tests with a fix category is to offer guidance to testers, directing them toward potential issues responsible for the flakiness. In this research, we first develop a framework utilizing pre-trained LMs to predict the fix category of a flaky test case solely based on the test case code. Our primary aim is to establish this prediction process as a black-box approach, solely reliant on the flaky test code without accessing the production code. Our study is different than existing work [4] whose focus is on predicting the category of flaky tests with respect to their root causes. Since knowing only root causes, especially at a high-level, does not necessarily help in fixing the test code, our work focuses on predicting fix categories in terms of what part of the test code needs fixing. Therefore, such fix categories are expected to guide the tester towards concrete modifications and thus decrease their manual investigation effort. Note that these categories can be useful for both

human developers and testers to manually fix flakiness and for tools to automatically and fully repair flaky test cases. In this work, however, we only report on the latter.

## 3.3 Approach

Our approach is divided into two phases: (a) automatically constructing a high-quality dataset of fix categories, based on existing datasets of flaky test cases and their fixes; (b) building a prediction model of fix categories, using the created dataset, to guide the repair of flaky tests.

### 3.3.1 Automatically Constructing a Dataset of Fix Categories

Having an appropriate and well-defined methodology for constructing a high-quality dataset of fix categories is important in practice, as we expect that local prediction models will need to be built in most contexts based on locally collected data.

In a nutshell, our approach starts with an existing set of fix categories and then extends it to a more comprehensive and practical set, and finally devise heuristics to automatically label fixes in existing flaky test datasets.

To the best of our knowledge, the categories of flaky test fixes presented by Parry et al. [87], as shown in Table 3.1, represent the largest and most recent classification in the literature. These fix categories were manually assessed by the authors by analyzing code diffs, developer's comments, and any linked issues of 75 flakiness-repairing commits that originated from 31 open-source Python projects [87].

Their categorization procedure accepts multiple sources of information (i.e., fixes but also comments and linked issues) to link a flaky test with a fix category. This information might not be entirely available for many flaky tests in a project. Therefore, following that procedure, not all fixes can be labeled with a category if we have only access to the flaky tests and their fixes, which is the case for many datasets.

In our research, we aim to categorize fixes and train a prediction model whose objective is to assign a fix category to a new flaky test, without any other other information than the test case's code. In other words, the goal is to guide testers (or automated repair tools) in applying the right fix according to categories of fixes that are predictable based on the code of the test cases alone. While not all flaky tests can be resolved solely by addressing the test code, a subset can indeed be fixed in this manner. Automating these fixes can still save time and resources. Hence, our dataset creation is targeting these specific fixes. To achieve that, we analyzed flaky test cases and their fixes, to extract ground truth labels. Inspired by Parry et al.'s work [87], we analyzed the International Dataset of Flaky Tests (IDoFT)[1], which comprises flaky test cases along with their fixes from 123 open-source Java projects.

[1] `https://mir.cs.illinois.edu/flakytests`

This dataset, utilized in various prior studies on flaky test case prediction [53–56, 103, 115], [27], is a publicly accessible dataset regularly receiving updates. We ensured our usage involved the most recent version of this dataset.

To assign labels to IDoFT dataset with fix categories, two researchers manually and independently analyzed 100 randomly selected flaky tests and their corresponding fixes to map them to one or more categories from those proposed by Parry et al. (see Table 3.1) or a new category when the fix did not match any of them. The new categories should be predictable by only looking at the flaky tests and yet be useful enough to help developers and testers fix them. Differences in categories and definitions between the researchers were resolved through multiple meetings. We use Cohen's Kappa [63] to measure the initial level of agreement between the two researchers. This is briefly discussed in section 3.4. There was no case where a third opinion was required to resolve a conflict.

Some of the fix categories proposed by Parry et al. are also renamed in Table 3.1 for better understandability. For example, the term *Reduce Scope* (one of the Parry et al.'s categories) lacks clarity in indicating the exact code section that requires modification. By renaming *Reduce Scope* to *Change Condition*, developers and testers are more explicitly instructed to modify conditional statements. The process resulted in 13 different fix categories for flaky tests, as shown in Table 3.2.

The final step in this phase was to create a set of heuristics to automatically label each fix from our dataset to one of these 13 categories. Each heuristic is a search rule (implemented as a Python script) that relies on keywords from the *diff* of a flaky test case and its fixed version. Subsequently, we manually verified the outputs of the automated labeling process to ensure the accuracy of the heuristics for future datasets. The complete labeling process is shown in Figure 3.1. Additionally, after this final step, the remaining 462 tests from our dataset were automatically labeled using the developed heuristics. Note that a single test may be labeled with multiple fix categories, as illustrated in Table 3.2.

In the following, we briefly explain each category's heuristic along with few examples from the IDoFT dataset.

For the **Change Assertion** category, the heuristic looks for the assert keyword and its replacements within the fixed flaky test code. For instance, if *AssertEquals*, *AssertThat*, or *AssertTrue* appear in deleted lines and *AssertJsonStringEquals*, *assertJSONEqual*, *AssertJsonEqualsNonStrict*, or *JsonAssert.AssertEquals* are present in the added lines, it shows an assertion change to address flakiness.

A common instance of flakiness arises due to the use of an incorrect assertion type. For example, *assertJSONEqual* should be used instead of *AssertThat* in an assert function that retrieves a JSON object from other functions in the test code and compares it with a hard-coded JSON String to ensure the deterministic ordering of the retrieved JSON object.

Changing data structures can be an effective technique to improve test reliability and reduce flakiness [68, 81]. For example, the data structures *Hash Map*, *Set*, *HashMap*, and *HashSet* are unordered collections. Their use in the test code can lead to test flakiness if the test relies on the order of elements in the structure. In such cases, these data structures should be replaced with *LinkedHash*, *LinkedMap*, *LinkedSet*, *LinkedHashMap* and

| Cause | Add Mock | Add/Adjust Wait | Guarantee Order | Isolate State | Manage Resource | Reduce Random | Reduce Scope | Widen Assertions | Miscellaneous | Total |
|---|---|---|---|---|---|---|---|---|---|---|
| Async. Wait | 1 | 6 | - | - | - | - | - | 2 | - | 9 |
| Concurrency | - | 2 | - | - | 2 | - | - | 2 | 2 | 8 |
| Floating Point | - | - | - | - | - | - | - | 3 | - | 3 |
| I/O | - | - | - | - | - | - | - | - | - | - |
| Network | 3 | 3 | - | - | 1 | - | - | - | 1 | 8 |
| Order Dependency | - | - | - | 2 | - | - | 1 | - | - | 3 |
| Randomness | - | - | - | - | - | 6 | - | 4 | 1 | 11 |
| Resource Leak | - | - | - | - | 2 | - | 1 | 1 | - | 4 |
| Time | 5 | - | - | - | - | - | 1 | 1 | 2 | 9 |
| Unordered Coll. | - | - | 3 | - | - | - | - | - | - | 3 |
| Miscellaneous | 2 | - | 1 | - | 1 | - | - | 6 | 7 | 17 |
| Total | 11 | 11 | 4 | 2 | 6 | 6 | 3 | 19 | 13 | 75 |

Table 3.1: Each flaky test is assigned a category describing the cause of flakiness (rows) and another describing the repair applied by developers (columns). From Parry et al. [87]

| Flaky Test Fix Category | Number of Flaky Tests |
|---|---|
| Change Assertion | 197 |
| Change Condition | 121 |
| Reset Variable | 104 |
| Reorder Data | 92 |
| Change Data Structure | 91 |
| Handle Exception | 52 |
| Miscellaneous | 50 |
| Change Data Format | 46 |
| Reorder Parameters | 37 |
| Call Static Method | 9 |
| String Matching | 6 |
| Change Timezone | 3 |
| Handle Timeout | 2 |

Table 3.2: Count of flaky tests within each fix category, using automated labeling. Categories with over 30 instances are highlighted.

*LinkedHashSet*, respectively, as they maintain the order in which their elements are stored, regardless of how many times a test is executed. This makes the test more reliable and less prone to flakiness. If *Hash*, *Map*, *Set*, *HashMap*, and *HashSet* keywords are replaced in the test code 'diff' with *LinkedHash*, *LinkedMap*, *LinkedSet*, *LinkedHashMap* and *LinkedHashSet*, respectively, then such test cases are labeled as **Change Data Structure**.

Exception handling is usually an important tool to reduce unwanted behavior due to wrong inputs or scenarios and thus decrease flakiness. If there are *Try Catch* statements added or removed or changes occur in the type of the Exception used in the test case, the test case is labeled as **Handle Exception**. Handling exceptions, similar to other fix categories, can be combined with different fix strategies to entirely eliminate the root cause of flakiness within a test case.

A flaky test is labeled with the **Reset Variable** category if one of the keywords *reset*, *clear*, *remove* or *purge* is found in the *diff*. These are useful commands to reset the state of the system and clear the memory at the end of the test cases. They put the system back to its initial testing state, which is required for many tests to avoid flakiness. Input formatting mismatches can be a cause of flakiness. Oftentimes, a fix is simply changing the input variable's format to the required one. If there is change in the format of the string or numeric data that is used in a function of the test code, the test case is labeled as **Change Data Format**.

If conditional statements are added or removed in the test code, for example replacing the *containsexactly* keyword with the *containsexactlyinanyorder* keyword, the test case is labeled with the **Change Condition** category. If there is any of the keywords *sortfield*, *sort_properties*, *sorted*, *order by* or *sort* in the fixed test code, which is used to sort or change the order of data, then the test is labeled as **Reorder Data**.

If there is a change in the order of the function parameters, then the **Reorder Parameters** label is used.

Tests may rely on specific dates or times, which can be affected by time zone differences, leading to inconsistent test results. Setting a specific time zone ensures consistent results, increasing test reliability and reducing flakiness [60] [128]. If there is a change in the assigned value to the *timezone* keyword, then such tests are labeled as **Change Time Zone**.

If the *Timeout* keyword is added to the test code, which ensures that tests are completed within a reasonable time frame, thus reducing the likelihood of flakiness, then these test cases are labeled as **Handle Timeout**. This fix has been suggested in the work of Pei et al. [90] to fix flakiness related to Async Wait.

Static methods can help ensure consistent test behavior across multiple runs by encapsulating complex scenarios or behavior simulations, such as network failures or time delays. If there is a static method call added to the test code, then these tests are labeled as **Call Static Method**. If two strings are matched in a given test to make sure they are in the correct order for maintaining determinism by using keywords like *'match.that(s).isequalto* or *.matches* then these tests are marked as **String Matching**. Lastly, if there is some change in the added or deleted lines of code that do not belong to any of the above cate-

gories, the test cases are labeled **Miscellaneous**. Examples of different fix categories are provided in our replication package [26].

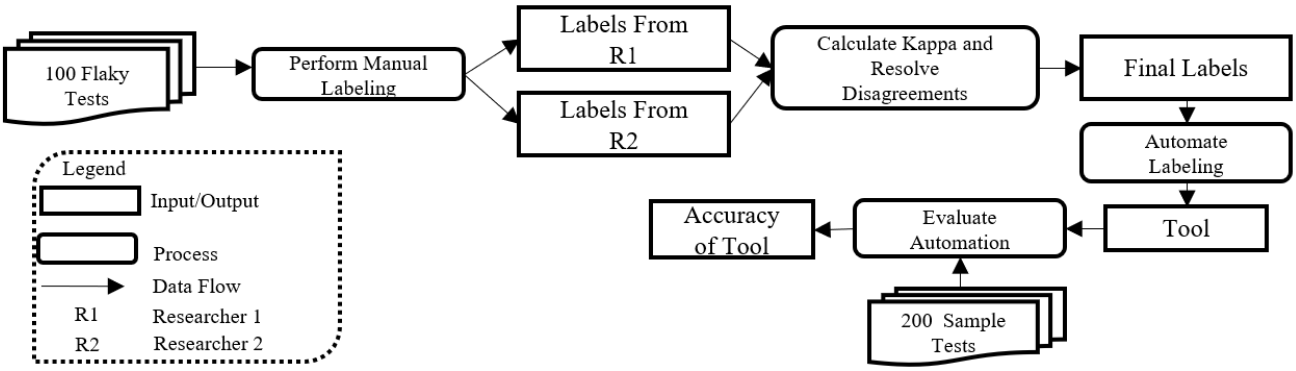



Figure 3.1: Automated labeling of flaky tests with their respective fix categories.

### 3.3.2 Flaky Test Case Fix Category Prediction with Code Models

In this section, we describe how we use different LMs for predicting the fix category of a flaky test. This prediction may be not only useful to help testers identify the required fix but can also guide any automated repair process.

We use two different code models CodeBERT and UniXcoder as our prediction models, respectively. In general, LMs are pre-trained using self-supervised learning, on a large corpus of unlabelled data. They are then further fine-tuned using a specific, labeled dataset for different Natural Language Processing (NLP) tasks such as text classification, relation extraction, or next-sentence prediction [29]. As briefly discussed in Section 2.3.1, Code models are those LMs that are specifically pre-trained on programming languages. CodeBERT and UniXCoder have been pre-trained using a diverse corpus encompassing code from six programming languages: Java, Python, JavaScript, PHP, Ruby, and Go. Both models excel in code classification tasks when compared to other pre-trained code models [27, 77]. These models comprehend both the syntax and semantics of various programming languages. We individually fine-tune these code models on our dataset of flaky tests to learn the underlying patterns of test code flakiness. During inference, these fine-tuned models classify flaky tests based on what these code models learned during training (fine-tuning). We choose not to fine-tune LLMs like GPT or CodeLlama for fix category prediction due to the high costs associated with their fine-tuning. In addition, as we will see in the results section the smaller code models will already provide very high accuracy which leaves no reason for us to switch to much more complex and heavier models. This process demands extensive data and substantial computational resources [127]. Reported studies show that smaller code models work well for classifying code [77] and are more cost-effective. Their efficient classification abilities, along with reduced time and computational expenses for refinement, make them more practical for integration into solutions for large-scale industrial systems.

For fine-tuning CodeBERT and UniXcoder to predict flaky test fix categories, we use two different techniques. In the first one, we augment the models with FSL using a Siamese Network to deal with the usually small datasets of flaky tests that are labeled in a typical development context. In contrast to FlakyCat, that also uses FSL [4], we train our model on the dataset created following the procedure presented in section 3.3.1, where each data item is a pair of flaky test and its fix category. Note that we do not use information from the actual fix during inference time, since our goal is to predict the fix category for a given flaky test which has not been repaired yet. Thus, in practice, only the test code is available to the model. In the second technique, we fine-tuned the models using a simple Feed Forward Neural Network (FNN) as done in earlier work for flaky test classification [27].

Furthermore, note that we do not build a multi-class prediction model to deal with multiple fix categories. Instead, we build multiple binary classification models to estimate the probability of whether a test belongs to each category. This is because flaky tests may belong to multiple fix categories. With multiple binary classifications, each fix category is treated separately, allowing the model to learn the features that are most relevant for that fix category. Since our dataset is relatively small, it is not computationally expensive to build these multiple classifiers for each category. However, as our dataset expands to include more samples, we may consider transitioning to a multi-class classifier for efficiency and scalability purposes in the future.

#### Converting Test Code to Vector Representation

Prediction using LMs starts with encoding inputs into a vector space. In our case, we embed test cases into a suitable vector representation using CodeBERT and UniXcoder, respectively.

Both CodeBERT and UniXcoder take the test case code as input and convert it into a sequence of tokens. These tokens are then transformed into integer vector representations (embeddings). The test case code is tokenized using Byte−Pair Encoding (BPE) [102], which follows a segmentation algorithm that splits words into a sequence of sub-words.

Each token is then mapped to an index, based on the position and context of each word in a given input. These indices are then passed as input to the CodeBERT and UniXcoder models to generate a vector representation for each token as output. Finally, for a test case, all vectors are aggregated to generate a vector called `[CLS]` [27].

#### Fine-tuning Code Models

As discussed earlier, we have fine-tuned CodeBERT and UniXcoder independently with two different techniques. Given our limited labeled training data, which is a common situation in our application context, few shot learning approaches [114], such as the Siamese Network architecture [40], are adequate candidate solutions. A Siamese Network trained with a specific loss function (*Triplet Loss Function* [101]) is designed to determine the similarity between inputs by learning an embedding space [75]—a high-dimensional vector space where inputs are represented as embeddings or vectors. In this space, similar inputs

are positioned closer together, while dissimilar inputs are placed farther apart. Similar to existing work [4], we constructed a Siamese Network [4, 48, 73] with three identical sub-networks. Each sub-network is based on either CodeBERT or UniXcoder, essentially using the same model (weights and network parameters) three times within the network structure. The inputs to this Siamese Network are *anchor*, *positive* and *negative* tests, which are transformed by each sub-network of the Siamese Network into high-dimensional embeddings [75]. The anchor and positive tests, which belong to the same class, are projected into the embedding space such that their distance is minimized, while the distance between the anchor and the negative test, belonging to a different class, is maximized. This process ensures that similar inputs are grouped together in the vector space, facilitating tasks such as similarity comparison or classification based on learned similarities. During training, the triplet loss function fine tunes the embeddings by computing similarity scores, encouraging the network to improve its ability to distinguish flaky tests belonging to different categories. In our approach, given a category (e.g., *Change Assertion*), the anchor is a flaky test in the training dataset that belongs to the positive class of that category, e.g., it is labeled as *Change Assertion*. A positive sample is another flaky test in the training dataset that belongs to the same class as the anchor (e.g., labeled again as *Change Assertion*) and is considered similar to the anchor. The negative sample is another flaky test from the dataset that belongs to a different class than that of the anchor (e.g., NOT labeled as *Change Assertion*) and is considered dissimilar to the anchor. More details about the procedure to create these samples in our experiment will be provided in Section 3.4.

We use the same architecture as Akli et al [4] for each sub-network. First, it includes a linear layer of 512 neurons. To prevent model over-fitting, we then add a dropout layer to randomly eliminate some neurons from the network, by resetting their weights to zero during the training phase [22]. Lastly, we add a normalization layer before each sub-network that generates a `[CLS]` vector representation for the input test code of that sub-network. Figure 3.2 depicts the architecture of a Siamese Network where anchor, positive and negative tests share a LM and other network parameters to update the model's weights based on the similarity of these tests. For training both CodeBERT and UniXcoder, we use the AdamW optimizer [124] with a learning rate of $10^{-5}$ and a batch size of two, due to available memory (8 gigabytes).

To evaluate the final trained model, following FSL, we create a support set of examples, for each prediction task. As a reminder, each prediction task here is a binary prediction of flaky tests indicating whether they belong to a fix category (positive class) or not (negative class). The support set includes a small number of examples for each class, from the training dataset. For every flaky test case in the support set of each class (positive and negative), we first calculate their `[CLS]` vector embeddings. We then calculate the mean of all test case vectors in that class, to represent the centroid of that support set class.

Each flaky test in the test data is referred to as a *query*. To infer the class of a query, from the test data, we first calculate the vector representation of the query using the trained model. Then using the cosine similarity of the query and the centroid of the classes, we determine to which class (positive or negative), the query belongs, i.e., the one with the highest similarity. Figure 3.3 shows the process of predicting the class of a query using FSL.

Regarding our second method for fine-tuning CodeBERT and UniXcoder, we train a Feedforward Neural Network (FNN) to perform binary classification for each fix category. This process is depicted in Chapter 2 in Figure 2.2 and was used by Flakify, which relied on CodeBERT [27]. The output of the LM, i.e., the `[CLS]` token which is the aggregated vector representation token of all input tokens, is then fed as input to a trained FNN to classify tests into the fix categories. The FNN contains an *input* layer of 768 neurons, a *hidden* layer of 512 neurons, and an *output* layer with two neurons. Like previous work [27], we used ReLU [3] as an activation function, and a *dropout* layer [124]. Lastly, we used the *Softmax* function to compute the probability for a flaky test to belong to a given fix category.

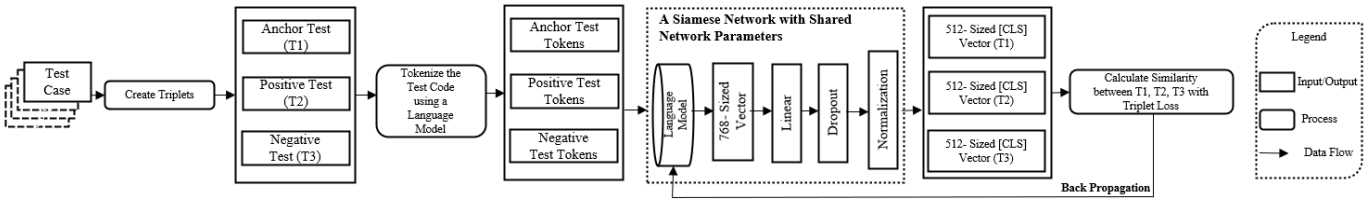



Figure 3.2: Training of a Siamese network with a shared LM and network parameters.

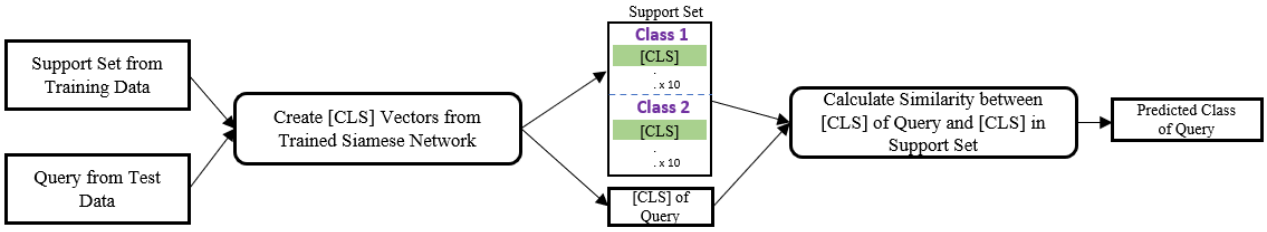



Figure 3.3: Predicting the class of query test case using few shot learning and a trained Siamese network.

## 3.4 Validation

The goal of this study is to predict fix categories for a flaky test to help generate a fully repaired version of the test. This section validates our approach (See Section 3.3) towards that goal by first introducing our research questions and explaining the data collection procedures. Then it discusses the experimental design and report the results.

### 3.4.1 Research Questions

In this study, we address the following research questions:

**RQ4** How accurate is the proposed automated flaky test fix category labeling procedure?
**Motivation**: Given that there is no existing dataset of flaky tests that is labeled by fix categories, in this RQ, we validate our proposed procedure to automatically label a flaky test given its fix. If accurately automated, such a procedure is quite useful since it can convert any flaky test dataset that includes fixes, to an appropriate training set for building prediction models, such as those used in RQ5. In practice, a customized training set (from local projects) is often necessary to capture domain knowledge, either to train new models from scratch or fine-tune pre-trained models, which further justifies the need for an automated labeling technique.

**RQ5** How effective are different LMs, i.e., CodeBERT and UniXcoder with and without FSL, in predicting the fix category, given flaky test code?
**Motivation**: Most existing prediction models related to flaky tests try to predict flakiness. FlakyCat [4] is, however, the only approach that classifies flaky tests according to their root cause using CodeBERT and FSL. Identifying the root cause alone, however, does not directly aid testers in resolving flakiness. For instance, one suggested root cause in FlakyCat [4] is *Unordered Collections*, highlighting that it causes flakiness. However, the proposed solution categories in [87] maps this cause to the *Guarantee Order* fix category without providing concrete guidance on how to ensure such order. In contrast, our defined fix categories offer a one-to-one mapping of specific fixes to individual test cases. For instance, concerning *Unordered Collections* cause, Table 3.2 presents multiple fix categories such as *Reorder Data*, *Change data structure* or *Reorder Parameters*. Then depending on the test code one of these concrete fixes is going to be predicted as the fix category. This approach provides testers with more precise solutions rather than a broad label like *Guarantee Order*.

This practical difference between a fix and a root-cause, led us to investigate alternative approaches for classifying the tests according to their required fix category, given that these categories are based on actual fix patterns. In this RQ, we explore two code models as representatives of the state of the art. We also study the effect of FSL with each model, since FSL is usually recommended when the training data set is small, like in our case.

### 3.4.2 Dataset and Data Augmentation:

To create a labeled dataset of flaky test fix categories for our study, we first analyzed various datasets of flaky tests that include fixes. Then we picked the most suited dataset and augmented it with our automatically generated labels.

#### Existing Datasets

There is no publicly available dataset of flaky tests that have been labeled based on the nature of their fix. Most existing datasets for flaky tests, such as FlakeFlagger [6], simply contain binary labels indicating their flakiness. The dataset presented in the most related work, FlakyCat [4], assigns a label to each flaky test based on the category of flakiness

causes, not their fixes. Therefore, we needed to build our own labeled dataset out of an unlabeled one that consists of both flaky tests and their fixes. We also required a dataset with a sufficient variety of flaky test fixes. Previous research on flaky test repair [20] has relied on datasets of limited size and scope. For instance, the Flex [20] dataset comprises only 35 flaky tests. Eck et al. proposed a study based on the Mozilla database of flaky tests [21] where they analyzed 200 flaky tests repaired by developers. Another example is the ShowFlakes dataset [87] that includes 75 commits from 31 Python projects. However, it only includes 20 flaky tests where flakiness can be removed by only changing the test code. The iPFlakies and iFixFlakies datasets [103, 113] are larger in size compared to previous datasets, but the root cause of flakiness in these datasets is related to the order in which the tests are executed, not the test code logic. The largest publicly available and relevant dataset is the International Dataset of Flaky Tests (IDoFT)[2], an open-source dataset that is continuously updated. In this research, we collected a Java dataset of flaky tests from IDoFT until October 23, 2023. The dataset originally contained 5,500 tests, including 128 duplicates. After removing duplicates, we were left with 5,372 tests. We then filtered this set to include only those flaky tests for which the fixes were accepted, meaning that the pull requests (PRs) addressing these flaky tests on GitHub were approved by the original developers, thus resulting in 788 tests. From this subset of flaky tests, we analyzed the diff of the original flaky tests and their developer repairs. We removed 226 tests that had repairs exclusively in the code under test (CUT) with no changes to the test code, leaving 562 tests that were fixed by modifying the test code. As explained in the Introduction section as well, the flaky tests that can be fixed in a black-box manner are in the minority, but they are still important since testers and testing tools can fix them even without any access to production code, which is typical for some QA teams. The remaining tests were resolved through test configuration adjustments, changes in test case execution order, fixes implemented in the production code, or are pending in open pull requests (awaiting approval, rejection, or being deliberately skipped by the original developers). More details about this can be found online on the IDoft Repository.

Consequently, our final dataset comprises 562 flaky tests and their corresponding resolutions from the IDoFT Java dataset. Notably, this dataset represents the largest available repository of flaky tests that are fixable through modifications of the test code. These tests originate from 123 distinct projects, offering a broad spectrum of test cases, which greatly benefits the generalizability of our predictive models.

### Labeled Dataset of Flaky Test Fix Categories

To create the labeled dataset out of the IDoFT dataset, we followed the procedure discussed in Section 3.3 and labeled each flaky test automatically. Our dataset is publicly available in our replication package [26]. The resulting labeled dataset is unbalanced with respect to the distribution of tests across fix categories. Though, in our prediction phase, to handle the lack of data, we apply Few-Shot learning, to effectively predict the fix categories will require a minimum number of test cases per category. Therefore, we selected the categories

[2] `https://mir.cs.illinois.edu/flakytests`

with at least 30 tests as our training set. As a result, our labeled dataset consists of the top nine fix categories outlined in Table 3.2 for training our model. Since a flaky test may be associated with multiple fix categories, we have transformed our multi-label dataset into nine distinct bi-labeled sets, one per category. In each of the nine sets, the positive class includes tests that belong to the corresponding fix category and the negative class includes all the other tests. This allows us to treat the problem of predicting each label (category) as a separate binary classification problem.

#### Data Augmentation for FSL

We investigated code models with and without using FSL. Without FSL, we fine-tuned the code models with an FNN as discussed in Section 3.3. For this, we used random oversampling [11], [27], which adds random copies of the minority class to our training dataset, which otherwise would be imbalanced (fewer positive examples).

In our fine-tuning approach, inspired by FlakyCat [4], we addressed limited training data by employing data augmentation techniques [104, 111] along with a Siamese Network, triplet loss function, and FSL. Specifically, we synthesize additional valid triplets for training, by creating <anchor, positive, and negative> tests for each flaky test in the training data, thus increasing the training samples. For example, if the original training dataset has 20 samples, data augmentation will potentially create 20 new positive samples and 20 new negative samples, resulting in a total of 60 training samples. For each triplet per category, the positive test must belong to the positive class (same as the anchor test), while the negative test must belong to the negative class. To synthesize additional positive and negative tests, we mutate the test code by replacing the test names and changing the values of variables (integers, strings, and Boolean values) with randomly generated values. For example, for a test case in the training set named *postFeedSerializationTest*, we create a new test case (and label it with the same class label) and call it *postFeedSerialization-Test_xacu4*. Then we change the value of an integer variable 'access point' from '1' to '8967', randomly. Also, the string value is randomly replaced from 'AMAZON-US-HQ' to 'AMAZON-"RZjqJ"-HQ'.

We automatically verify the created pairs, to ensure no duplicate tests are generated and to retain all flakiness-related statements unchanged. We thus produce enough samples to form the triplets (a set of anchor, positive and negative tests) required for this fine-tuning procedure.

### 3.4.3 Experimental Design

This subsection covers the baselines, code model training and testing, GPT model's prompt template, and execution of the repaired flaky tests.

### Baseline

Automatically defining different categories of flaky test fixes using only test code and then labeling flaky tests accordingly along with generating a full repair of flaky tests is a task that has not yet been studied before. Previous work has either analyzed the causes of flakiness [4] or manually identified the fix for flaky tests [87]. To design our experiment, we only focused on code models for prediction, as explained in Section 3.3.2. As our technique relies exclusively on test code, we cannot compare it with traditional machine-learning classifiers, which require a pre-defined set of features such as execution history, coverage information in production code, and static production code features [27]. Alternatively, only relying on predefined static features from test code is challenging. These features may not capture all the semantics of the code that contribute towards flakiness and may not be good indicators to predict fix categories. Lastly, defining good features, often referred to as feature engineering [111], is not an easy task. In contrast, we aim to develop a solution that is practical where testers just pass the test code directly to a model as input and get information regarding their fix category, rather than going through the process of defining features.

We excluded traditional sequence learning models as baselines, such as the basic RNN or LSTM models, since they are not pre-trained like code models. Hence, they require large datasets to train their many parameters and avoid over-fitting [17, 88].

Therefore, we evaluated our idea using two SOTA code models, namely CodeBERT and UniXcoder, both with and without FSL. FSL has been recommended in previous work for small datasets (e.g., for flakiness root-cause prediction [4]), but has not been compared with code models without FSL. Thus we also included this experiment.

### Training and Testing Prediction Models

To evaluate the performance of our code models for predicting the fix categories, we employed a stratified 4-fold cross-validation procedure to ensure unbiased train-test splitting. This procedure involves dividing the dataset into four equal parts, with three parts allocated for training the model and one part held out as the test set. Moreover, to prevent over-fitting and under-fitting during the model's training, we employed a validation split, which consists of 30% of the training data. This validation dataset is used for fine-tuning code models using FNN and a Siamese Network. We deliberately did not use a higher number of folds since for smaller categories that would not leave enough samples in the train and test sets. For instance, the *Change Data Format* fix category has only thirty-three positive examples, which would lead to only four positive examples in each fold of a 10-fold cross-validation.

### Evaluation Metrics

To assess the effectiveness of our classifier, we employed standard evaluation metrics, including precision, recall, and F1-Score, which have been widely used in previous studies

of ML-based solutions for flaky tests [4, 6, 14, 27, 93]. In our case, precision is of utmost importance, and we emphasize its role in our results. In practical terms, once a fix category is predicted for a flaky test, such as *Change Assertion*, we want testers to confidently proceed with fixing the issue by focusing on the code statements matching the category, e.g., by modifying the assertion type used. Accurately predicting the correct fix category is therefore critical since attempting to fix a test incorrectly can result in wasted time and resources [69] and may even cause additional failures in the test suite [37, 108].

We used Fisher's Exact test [97] to assess whether there was a significant difference in performance among the four prediction methods: fine-tuned CodeBERT and UniXcoder with and without FSL, across all nine fix categories. Furthermore, in RQ4, we used the Kappa score [63] to evaluate the level of agreement between the researchers who initially labeled the dataset. By using the Kappa score, we were able to assess the reliability and consistency of the initial labels across different categories of fix, despite their varying sample sizes.

### 3.4.4 Results

#### RQ4 Results

To start, 100 test cases were randomly selected and independently labeled by two researchers (that are among the co-authors of this research).The Kappa score was found to be 0.65, indicating a moderate level of agreement between the two raters. Disagreements betwe en them were eventually resolved with multiple rounds of discussions to finalize the fix categories that are used by the automated labeling tool.

To evaluate the labeling accuracy of our automated tool, we took another random set of 200 flaky tests and obtained their fix category labels from the tool. The same two researchers then manually labeled these tests, enabling us to calculate the accuracy of our automated labeling tool as shown in Figure 3.1. We found that the tool had an accuracy of 98.5%, with only three tests being mislabeled with multiple fix categories, including the correct one.

#### RQ5 Results

Table 3.3 presents the prediction results for all nine categories of flaky test fixes using four distinct approaches, namely CodeBERT and UniXcoder, fine-tuned with Siamese Networks or FNN. In the former case, we evaluated both LMs using Few-Shot Learning (FSL).

UniXcoder without FSL demonstrated superior precision and similar recall in the *Change Assertion*, *Change Data Structure*, *Handle Exception*, *Change Condition*, and *Reorder Parameters* categories, with precision of 96%, 89%, 98%, 87%, and 91%, respectively, outperforming the other three approaches. For *Reorder Data*, again UniXcoder without FSL achieves the highest precision score of 96%. However, when incorporating FSL, UniXcoder demonstrates a superior recall of 88%. Furthermore, CodeBERT without FSL showed

slightly better results for the *Miscellaneous* category, with 83% precision and 67% recall, though the difference with UniXcoder is not significant. For the *Reset Variable* category, UniXcoder with FSL yielded high accuracy with 96% precision and 97% recall rates, while CodeBERT without FSL achieved a slightly higher recall of 98% albeit with a low precision of 88%. Finally, for *Change Data Format*, CodeBERT with FSL outperformed all other approaches with a 97% precision and recall. However, again the difference with UniXcoder is not significant.

The *Change Data Format*, *Handle Exception*, *Reset Variable*, *Reorder Data* and *Change Assertion* categories were the easiest for all four approaches to classify, with an average precision score (across all approaches) of 93%, 96%, 92%, 88%, and 90%, respectively. This is likely due to the high frequency of common keywords in test code for exception handling, assert statements, resetting variables, and reordering data (with keywords like "sort" and "order by"). In the case of the *Change Data Format* category, the model looks for a long complicated string that is being passed to a function. These strings are easier for the model to identify because we observed that all the flaky tests matching this category contain a long complicated string that needs to be updated to remove flakiness. All four models demonstrate strong performance across most fix categories, exhibiting high recall and precision. However, in the Miscellaneous Category, their performance notably decreases, averaging at 62%. This lower performance may be attributed to the absence of specific common patterns or code keywords across all tests within this category, which cannot be expected to be homogeneous. Moreover, in the Reorder Parameter Category, CodeBERT, with and without FSL, exhibits significantly lower precision compared to UniXcoder. Further details about the models' misclassifications are provided in Section 3.5.

Overall, the results presented in Table 3.3 suggest that, for most of the fix categories, UniXcoder outperforms CodeBERT significantly, based on a Fisher's Exact test with $\alpha = 0.05$. However, FSL did not significantly improve UniXcoder's performance, which is probably due to the limited dataset with only a small number of positive instances for most fix categories, which hindered fine-tuning.

**RQ5 summary:** UniXcoder outperformed CodeBERT for predicting most of the fix categories, with higher precision and recall. The use of FSL does not significantly improve the prediction accuracy of UniXcoder.

## 3.5 Discussion

In this section, with qualitative analysis, we aim to ascertain not only theoretical soundness but also the practical utility of our solution in addressing the repair of flaky tests.

### 3.5.1 Qualitative Analysis of the Model's Performance

To gain a deeper understanding of the best model's performance, we have conducted a qualitative analysis of representative flaky tests from various fix categories.

| Metric | **Precision** (%) | | | | **Recall** (%) | | | | **F1** (%) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Category | CB | CB with FSL | UC | UC with FSL | CB | CB with FSL | UC | UC with FSL | CB | CB with FSL | UC | UC with FSL |
| CA | 88 | 86 | **96** | 93 | 88 | **86** | **86** | 85 | 88 | 86 | **91** | 89 |
| CDS | 86 | 76 | **89** | 85 | 86 | 71 | **87** | 81 | 87 | 74 | **88** | 83 |
| HE | 97 | 95 | **98** | 95 | 87 | 88 | 89 | **90** | 92 | 91 | **93** | **93** |
| RV | 88 | 91 | 94 | **96** | **98** | 95 | 97 | 96 | 93 | 93 | **96** | **96** |
| CC | 91 | 84 | **87** | 74 | 84 | 70 | **87** | 70 | **87** | 76 | **87** | 72 |
| RD | 88 | 80 | **96** | 90 | 82 | **88** | 72 | **88** | 85 | 84 | 82 | **89** |
| CDF | 96 | **97** | 90 | 90 | 93 | **97** | 94 | 90 | 96 | **97** | 93 | 95 |
| MC | **83** | 79 | 82 | 76 | **67** | 61 | 62 | 61 | **74** | **70** | 70 | 68 |
| RP | 53 | 48 | **91** | 88 | **94** | 88 | **91** | 82 | 64 | 68 | **91** | 85 |

Table 3.3: Comparison of the prediction results for each fix category using CodeBERT (CB) and UniXcoder (UC), both with and without FSL. The highest value per metric for each fix category is highlighted.

### Code Model's Prediction Results for Flaky Test Fix Category

In this section, we are analyzing sample test cases and their predictions using one approach, namely UniXcoder without FSL, since on mean it performs the best among the prediction models. We specifically target True Positives (TP) and a False Negatives (FN) from the same category, to provide the reader with insights regarding how these models can predict flaky fix categories. We also investigate False Positives (FP), since FP instances waste testers' time when they follow wrong suggestions and therefore directly affects the applicability of the proposed approach.

Figure 4.9 shows a representative example test whose fix category is correctly predicted as *Change Data Structure* (TP). On line 2 of this test code, **HashMap** is used, which can cause flakiness, as discussed in Section 3.3. Our model correctly predicts the category and suggests replacing **HashMap** with **LinkedHashMap**. Similarly, Figure 4.10 (a) shows another Flaky test whose fix category is correctly predicted as Handle Exception and Change Assertion (TP). Figure 4.10 (b) shows the developer fix of the same flaky test where a correct exception handling was added and *assertEquals* was changed to *assertJSONEqual* for correctly handling JSON objects. The test code shown in Figure 4.11 (a) is an example where our prediction model fails to correctly predict the fix category (FN) as *Handle Exception*. This test case differs from the typical patterns observed in that fix category because the keyword exception was already present in the method signature. Typically, exception handling is not part of flaky code in that category and an exception or try-catch is added in the code to resolve flakiness. However, in this case, it is the opposite and the pattern does not match other examples in the training set. When we examine the corrected test code shown in Figure 4.11 (b), we notice the exception had been removed as part of the fix. However, our automated labeling tool classified this test case as *handle exception* due to the presence of the 'exception' keyword.

In general, we do not have many examples of FP cases (13 tests on mean for all of the four approaches we have used), which is not surprising as it is one of the strengths

of our proposed solution. One sample FP test code is shown in Figure 4.12. This test was incorrectly classified as a *Change Assertion* since no change in the assertion type is needed. A potential reason for the misclassification can be related to the long string passed to the assert statement in the code. The pattern of using the long string is similar to cases where flakiness is addressed by changing an assertion to a more customized assertion with shorter inputs, for example using *AssertJsonStringEquals* to handle Json Strings. However, here the incorrect use of assert was not the cause for flakiness and thus was not changed. Although the recommendation was a FP, suggesting a Change in assertion might not be a bad idea after all, not as a fix but as refactoring to create more maintainable tests.

To summarize, we can conclude that the models are correctly learning the patterns that exist in the test code and, in most cases, FP or FN are simply due to lack of data, where without the actual fix or other sources of information about the bug, correct prediction is not possible.

## 3.6 Threats to Validity

### 3.6.1 Construct Threats

In our study, the main potential construct validity threat is the possibility of defining incorrect or incomplete labels in RQ4, which can undermine results in RQ5. To reduce this potential threat, we first made the process of labeling systematic and automated with heuristics that are publicly available in the replication package for further evaluation and improvement. In addition, we also manually checked random samples of the labeled tests, to confirm the correctness of the process.

Additionally, For RQ5, We did not conduct a per-project analysis in our study due to the limited number of positive instances for all categories of fixes and the large number of projects in our dataset (96). This implies that for each flaky test in the test set, very few flaky tests from the same project are used in the training set, especially for a specific category. We reported overall results across all projects.

### 3.6.2 Internal Threats

Internal threats to validity are about to make sure the cause-effect relation identified in the study is really there and there is no other explanation (confounding factors). In our study, while predicting the labels is the models context size (510 tokens for CodeBERT and 1022 for UniXcoder), which may truncate the source code of some test cases and result in information loss. In our dataset, 49 tests (11% of the total 562 flaky tests) exceed the token size limit of 510 tokens, and 17 tests (4%) exceed 1022 tokens. This could potentially impact the accuracy of our prediction results, especially for those fix categories where we have limited positive examples. Further research should investigate the extent to which this limitation may affect the outcomes of our study.

### 3.6.3 Conclusion Threats

To reduce potential conclusion validity threats, in RQ5, we used Fisher's Exact test to make sure our conclusions are statistically significant.

### 3.6.4 External Threats

Our automated labeling tool and the heuristics are defined for Java tests. We have not evaluated our tool on other programming languages but it can be adapted to other languages and new projects by incorporating adapted keywords and rules. Additionally, CodeBERT and UniXcoder models are pre-trained on multiple programming languages.

## 3.7 Conclusion

In this chapter we discussed our approach for categorizing flaky tests according to practical fix categories based exclusively on test code. Given the limited availability of open-source datasets for flaky test research, our generated labeled data can also be valuable for future research, particularly for ML-based prediction and repair of flaky tests. We evaluated two LMs, CodeBERT and UniXcoder, with and without Few-Shot Learning, and found that the UniXcoder model outperformed CodeBERT in predicting most fix categories.

# Chapter 4

# Repair of Flaky Tests

This chapter addresses $TO_3$, i.e. automatic repair of flaky tests using LLMs. This section is an extension of the work from Chapter 3. The contents of this chapter have been published as Research Question *RQ3* for publication in the Journal of *Transactions on Software Engineering (TSE)* [28].

## 4.1 Overview

This section is a continuation of the work we presented in Chapter 3. After predicting the fix category labels for the given flaky tests, we evaluate the performance of recent LLMs, namely GPT [80], which possess a vast number of model parameters (up to 175 billion in GPT 3.5 Turbo), in analyzing their capability to generate fixes for flaky tests using the predicted fix category labels. This is shown in Figure 4.1. Our findings indicate following key points: (1) Our predicted fix category labels play a significant role in guiding GPT to generate better full fixes for flaky tests. We evaluated this using CodeBLEU [19] and achieved a mean score of 81% when using predicted fix category labels in the input prompt, compared to 75% without them. These results are further enhanced with in-context learning [58], which we tested on three different fix categories; (2) Based on the execution and analysis of a sample of GPT-repaired tests, we estimate that a large percentage can be expected to pass, roughly between 46% and 80% of them. For the generated tests that fail, the required changes to correct them are limited, with an average of 16% of the tokens needing modification in the test code. The rest of this section is structured as follows. Section 4.2 discusses our flaky test repair approach in detail. Section 4.3 presents the evaluation of our results with different types of input prompts to the GPT model. Section 4.4 discusses the incorrect generations by the GPT model. Section 4.5 explains threats to the validity. Section 4.6 reports related work and finally, Section 4.7 concludes this chapter.

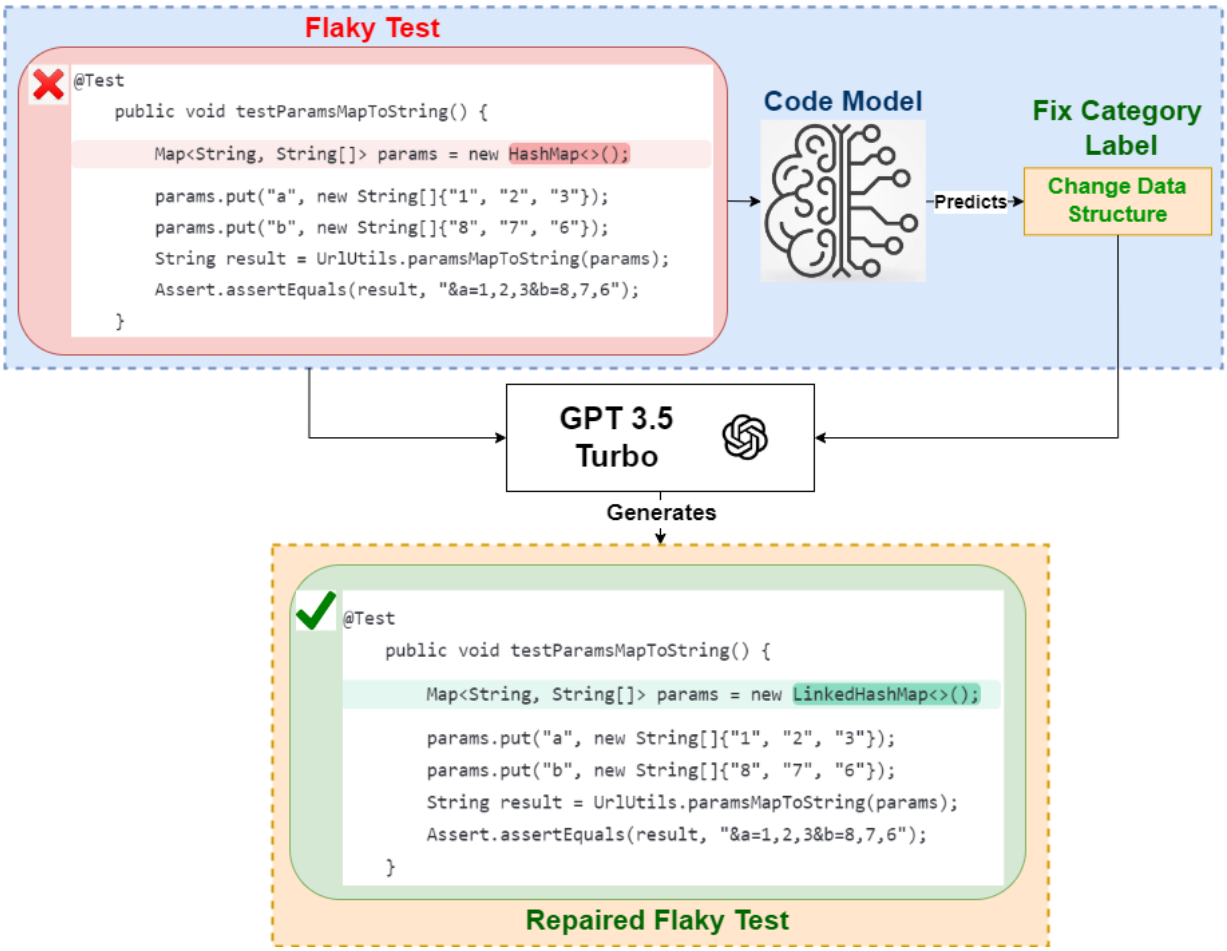



Figure 4.1: Predicting fix category label and generating repaired flaky tests using LLMs

## 4.2 Approach

This section presents our approach for generating repaired flaky tests using LLMs with the help of our predicted fix categories.

### 4.2.1 Repairing Flaky Tests Using GPT and our Proposed Fix Categories

LLMs such as GPT have been successfully applied to many automated software engineering tasks in the past [24, 129], including automated program repair [89, 116, 126] and [30]. Therefore, as part of our study, we are interested to see (a) how effectively an off-the-shelf GPT can generate fixes for our flaky tests and (b) how our classification of fixes can potentially help GPT to better fix flakiness.

We use GPT-3.5 Turbo, with 175 billion parameters and a context window of 16,385 tokens, designed to accommodate long inputs (in our case full test cases). This GPT version was released on November 06, 2023 and has shown superior performance for diverse code-

related tasks compared to prior versions. As per OpenAI's official website [79], this model has been trained on data up to September 2021.

GPT-4, which was unveiled early 2023, features 1.7 trillion parameters, significantly surpassing GPT-3.5 in performance. However, the latest GPT-4's training dataset extends up to December 2023. Further, there is another version of GPT-4 that is trained up to April 2023. Given that a majority of flaky tests in our dataset were posted on the public repository before these two dates, we chose not to use GPT-4 as it could lead to data leaks. Though there are more recent test case fixes in our dataset, most of them are not yet accepted changes and we only consider fixes that are approved by the developers.

#### GPT Prompt for Fixing Flaky Tests

To evaluate the impact of predicted fix categories on GPT's repair capability, we incorporate the predicted category alongside the flaky test code as input query to GPT (the prompt). We use two different methods to build the prompt. The first simply provides flaky test code with its corresponding fix category label, while the second employs in-context learning. Influenced by recent research on prompt optimization for GPT [58], in-context learning provides multiple examples of inputs and their expected outputs, as the context. Since fine-tuning a LLM is expensive, with in-context learning, the LLM learns at inference time [117] on how to optimize its output for a particular problem domain (characterized by the examples in the context).

Section 4.3.4 shows our both prompt templates. In our current implementation of in-context learning, a prompt contains the flaky test requiring repair, its predicted category, and some sample flaky tests from the same category in the training set, accompanied by their respective fixes. This methodology enriches the model's output (the fix) by exposing it to prevalent resolution strategies applied to flaky tests within a specific category. Future investigations could delve into further optimizing these in-context learning prompts [96] to get the best results possible from GPT.

## 4.3 Validation

### 4.3.1 Research Questions

In this section, we address the following research question:

**RQ6** How effective is GPT-3.5 Turbo in repairing a given flaky test, with and without information regarding its fix category and fix examples?
**Motivation**: Given a fix category, the next question is to check to what extent having that information helps in fixing flakiness. To answer that, we focus on a fully automated flaky test repair approach and want to assess whether knowing the flaky test category help obtain better repairs. To automate the repair process, we chose to rely on LLMs that

are pre-trained on an extensive corpus coming from various resources, enabling them to generate complete code without requiring fine-tuning.
The recent development of LLMs with 175 billion parameters for GPT-3.5 Turbo and 1.7 trillion parameters for GPT-4 [79], trained over a dataset size of 45TB [12] offers significant potential for automating tasks related to source code, including automated program repair [30]. Thus RQ6 investigates whether the fix categories predicted in RQ5 enhance the performance of GPT models in repairing flaky test cases and to what extent we can achieve correct repairs. Additionally, we use in-context learning to understand how the GPT model's ability to fix issues improves when given examples of a specific type of flaky fix alongside our suggested fix category label.

### 4.3.2 Data Selection for Evaluation with GPT Model

In RQ6, we selected 562 tests to assess how well the GPT model deal with our flaky tests with and without fix category. Since the GPT 3.5 model was trained on collected data up to September 2021, we only included tests that were fixed or added after that date. Including older tests could be considered data leakage, since these tests are in the public domain (GitHub) and there is a high chance that they are in the model's training set. This exclusion criterion resulted in 181 tests from 61 distinct projects which included at least one test from each of the nine fix categories.

### 4.3.3 Data Selection for Execution of Repaired Flaky Tests

To gain further insights into the GPT-generated repaired flaky tests, we conducted executions of selected tests. Our dataset comprises 181 tests from 61 distinct projects. Among these projects, we selected projects where at least five flaky test cases were present, which resulted in nine projects. Out of these nine projects, the test cases of four projects could not be executed due to missing dependencies, resulting in build failures, or the unavailability of certain files linked to flaky tests mentioned in the IDOFT dataset, either due to merged commits or non-existent files in our local system. Consequently, from these 5 successfully executed projects, we collected results of all their flaky tests (35 test cases).

### 4.3.4 Experimental Design

This subsection covers the GPT model's prompt template and execution of the repaired flaky tests.

#### Prompt Design

To get the fixed flaky test code from GPT, we use three different types of prompts, as follows: (1) we pass the flaky test code in the prompt to the model and ask it to generate its fix as shown in Figure 4.2, (2) we provide both the flaky test code and the fix category

label we predicted for that flaky test. We then ask the GPT model to generate a fix as shown in Figure 4.3, and (3) we extend the content of the second type by also passing a few examples to the model as shown in Figure 4.4. This type of prompting aims to assess GPT's performance with in-context learning. We focused on three fix categories comprising more than 30 instances (to have large enough sample sizes to make meaningful conclusions) within our dataset of 181 tests: *Change Assertion* (64 examples), *Change Data Structure* (41 examples), and *Change Condition* (41 examples). This resulted in a dataset of 146 tests. From these, we randomly selected 5 examples from each category for in-context learning, leaving a total of 131 tests for our in-context learning experiments.

We used these selected examples of flaky tests, along with their respective fixes and fix category labels, as prompts for the GPT model. Our prompt requested the model to provide a fix for a new flaky test based on a given fix category label, utilizing information derived from these example tests and their associated fix category labels. Our prompt explicitly requests the model to provide only the fixed test code, excluding any additional textual descriptions. This approach streamlines the output, ensuring that the model generates the test code directly without requiring post-processing. In the future, we plan to further test our GPT model using a larger number of examples of flaky tests. This expansion aims to evaluate and enhance the model's capabilities in handling varied scenarios.

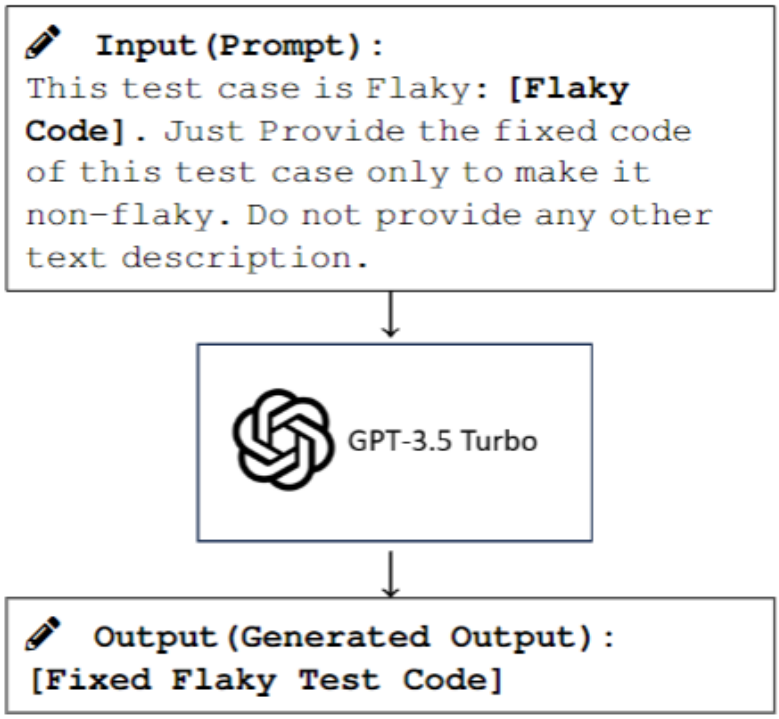



Figure 4.2: Flaky test fix using GPT-3.5 Turbo without fix category label.

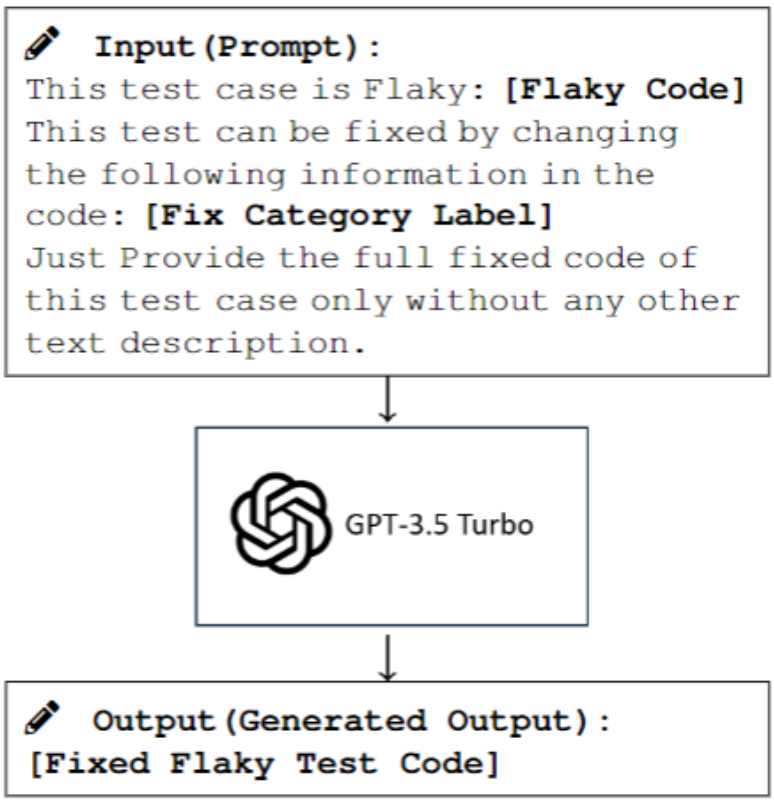



Figure 4.3: Flaky test fix using GPT-3.5 Turbo with fix category label.

### 4.3.5 Test Case Execution

To verify the automatically generated repairs, as described in Section 4.3.3, we ran 35 tests obtained from 5 projects that could be executed. To do so, initially, we executed the original developer-repaired versions of the 35 tests, 10 times each, to help verify the approved fix was indeed correct. Then, we executed the LLM-generated versions of these tests, 10 times each as well. Given that a test may remain flaky even after repair if the correction is inadequate, multiple runs were necessary to ascertain consistent outcomes across executions. That is, a passing or failing test should consistently pass or fail, respectively, in all 10 runs.

### 4.3.6 Evaluation Metrics

To evaluate how useful our classification is for improving an automated repair model (i.e., GPT in this case), we have used several metrics from the BLEU score family [84] (i.e., Corpus BLEU, Sentence BLEU, and CodeBLEU), which are widely used metrics in automated code generation and repair studies [2, 19, 57, 62].

Particularly, CodeBLEU is crucial as it captures both the semantics and syntax of the code, providing a more precise evaluation of the generated code. Unlike the traditional BLEU score, which assesses the similarity between candidate and reference texts, based

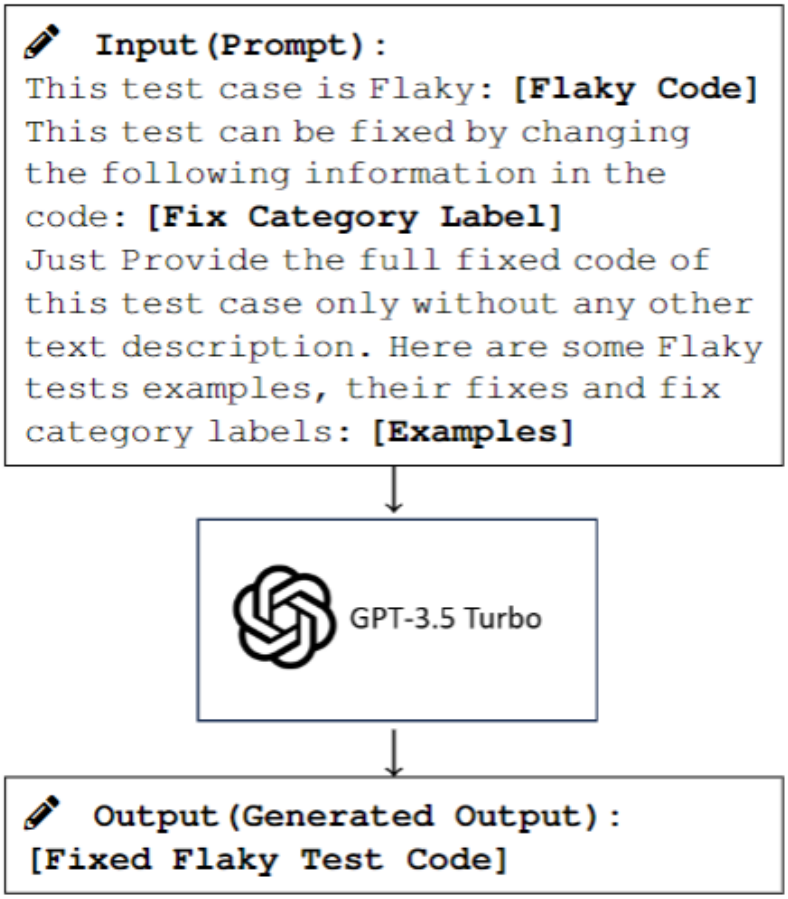



Figure 4.4: Flaky test fix using GPT-3.5 Turbo with in-context learning.

solely on sequences of $n$ contiguous tokens (n-grams) overlapping, CodeBLEU incorporates additional components to account for the specific code characteristics. It is defined as a weighted combination of four parts [98] as shown in the below equation.

$$\begin{aligned} \text{CodeBLEU} = \alpha \cdot \text{BLEU} + \beta \cdot \text{BLEU}_{\text{weight}} \\ + \gamma \cdot \text{Match}_{\text{ast}} + \delta \cdot \text{Match}_{\text{df}} \end{aligned} \tag{4.1}$$

where:

- BLEU represents the standard BLEU score, measuring the overlap of n-grams between the generated and reference code.
- $\text{BLEU}_{\text{weight}}$ denotes the weighted n-gram matching, assigning varying weights to n-grams to emphasize important code keywords like *int*, *char*, *public*, and others. Different weights are assigned to different n-grams, with keywords often receiving higher weights.
- $\text{Match}_{\text{ast}}$ signifies the syntactic AST match, focusing on the syntactic structure of the generated code.

- $\text{Match}_{\text{df}}$ assesses the semantic similarity between generated and reference code by examining their data-flow representations. Code is represented as a graph where nodes represent variables, and edges indicate how variable values flow. Unlike Abstract Syntax Trees (AST), data-flow graphs illustrate how variables interact within the code. We are thus able to measure the semantic match between the candidate and reference code.

The parameters $\alpha$, $\beta$, $\gamma$, and $\delta$ in Equation 4.1 represent the weights of the above-mentioned four parts in the CodeBLEU score. We have used a value of 0.25 for all four weights, as used by Ren et al. [98]. These four components collectively evaluate both the grammatical correctness and logical correctness of the generated code, making CodeBLEU a comprehensive metric for code evaluation.

Beside syntactic similarity metrics like CodeBLEU, we also executed a subset of repaired tests to assess the correctness of GPT-repaired flaky tests. Analysing the execution results, we calculated the passing rate of the repaired tests. Then using bootstrapping [72] with a 95% confidence level, a statistical method for estimating properties of a large sample from repeated sampling (with replacement) from a small sample, we established lower and upper bound percentages of passing test cases within the entire flaky test set, including non-executable test cases. Additionally, we used a logistic regression [42] model to predict the pass rates of non-executable tests based on their CodeBLEU scores in various prompt settings.

Furthermore, for passing tests, we manually ensured they did not introduce regressions regarding the test class's bug-finding capability. While tests with a CodeBLEU score of 100% didn't need equivalence checks, for those with scores below that, we manually verified behavioral equivalence with the ground truth. This was possible given the relatively low complexity of the test cases.

Regarding failing tests, we calculated the Levenshtein Edit Distance [100] between the generated and actual fixes. This is a measure of similarity between two strings that determines the minimum number of token edits needed to transform one into the other. We can then compute the number and percentage of tokens needing change to achieve perfect alignment with the original developer-repaired test.

### 4.3.7 Results

Table 4.1 reports on the evaluation results generated by GPT 3.5 Turbo, which processes a flaky test and provides its corrected version, with and without our predicted fix category labels as input. The assessment metrics include median and mean scores for Corpus BLEU, Sentence BLEU, and CodeBLEU across the 181 flaky tests in the test set. The inclusion of our predicted fix category labels, in the prompt, increases the median of all three metrics by 7%, 6%, and 4% for Corpus, Sentence, and CodeBLEU, respectively. The improvements are statistically significant based on non-parametric Wilcoxon signed-rank tests at a significance level of $\alpha = 0.05$, with a p-value below 0.0001 in all three cases. This improvement is attributed to the additional information provided to the GPT model, aiding it in precisely

identifying and addressing the issues causing test flakiness. This finding also indirectly suggests that our predicted fix categories are well defined and may effectively help testers identify the specific code segments that require modification to mitigate flakiness. In the rest of this RQ, we only report CodeBLEU scores, since they are designed to more directly assess the semantics and syntax of code rather than natural language text, thus providing a more accurate evaluation of generated test code.

Table 4.2 shows median and mean CodeBLEU scores per fix category, with and without using label information. As we can see, in most categories, the labels improve the mean and median CodeBLEU scores. The improvements per category range from 0% to 17% for the mean and from -1% to 22% for the median CodeBLEU. Smaller improvements are either in categories with very few samples (e.g., Reset Variables with only one sample) or when the results are already quite high (e.g., Change Condition with a median CodeBLEU of 89% without labels). The only category where the Median results slightly dropped (1%) after providing the labels is the Misc category. This is most likely because the *Misc* label does not offer practical information on how to resolve the flakiness.

Now to expand the analysis to the third prompting approach (in-context learning), Table 4.3 compares the three approaches for the *Change Assertion*, *Change Data Structure*, and *Change Condition* fix categories. As explained, we only look at these categories since after removing 5 samples per category to be included in the context, only these categories have enough samples to do a proper analysis and run statistical tests.

Looking at the table, we see that, for the *Change Data Structure* category, the mean and median CodeBLEU scores increase from 80% to 86% and 87% to 94%, respectively, when we use in-context learning versus simply providing the labels. Similarly, for the *Change Condition* category, mean and median CodeBLEU scores increase from 88% to 92% and 90% to 97%, respectively. However, in the case of the *Change Assertion* category, there is no discernible improvement in the model's performance after exposure to a limited number of examples. This could be attributed to the variability of assertion statements within the test code, suggesting that multiple approaches may exist to fix assertions and resolve flakiness. It is plausible that the number and variety of examples provided was insufficient for the model to grasp the diverse spectrum of assertion-related fixes. Further, as depicted in the boxplots in Figure 4.7, the overall distribution of the CodeBLEU scores shows significant variance for all types of prompts and the three fix categories. The low minimum values in all cases shows that there are flaky tests that are not fixed regardless of the prompting strategy. However, when looking at the CodeBLEU over all three categories for in-context learning, it is clear that, for the majority of the cases, the CodeBLEU scores are quite high. For example, even for the Change Assertion category, which has the lowest minimum scores, 75% of samples have CodeBLEUs higher than 55%, and for half of the test cases the CodeBLEU scores are greater than 85%.

Given the variance in CodeBLEU values per category, we again used the non-parametric Wilcoxon signed-rank test with a significance level of $\alpha = 0.05$ to make sure the improvements we see in medians are statistically significant as well. Our findings indicate that, when accounting for all three categories, both for prompts with label and without label, in-context learning makes a significant difference, with p-values below the significance

threshold $p < 0.001$. However, looking at individual categories, this difference for prompts with label is only significant for *Change Data Structure* and *Change Condition*, with p-values $p < 0.001$ and $p = 0.007$, respectively.

In future investigations, we intend to systematically explore the use of different prompts by augmenting the number and types of examples or experimenting with other prompt engineering strategies [57]. Further details regarding GPT's performance on individual fix categories are discussed in Section 4.4.

To gain deeper insights into the performance of GPT-repaired flaky tests, we conducted a series of evaluations based on test execution results. For the reasons explained in Section 4.3.3, we could only execute a total of 35 tests, each repeated 10 times, resulting in 24 passing and 11 failing tests, with consistent results across the 10 runs.

From the 35 executable tests, generated with labels and examples, for the three categories where the latter are available, we would like to first estimate the passing rate of fixed tests while accounting for the uncertainty associated with such a limited sample. Using Bootstrapping, we estimate a 95% confidence interval for the percentage of passing tests, that ranges from 51% to 83%. This interval is informative for testers, who do not have access to CodeBLEU scores in practical testing scenarios, and would like to know what to expect from FlakyFix. Based on these results, we conclude that FlakyFix provides an effective solution in a large proportion of cases.

Next we are interested in determining the relationship between execution outcomes and the corresponding CodeBLEU scores to ascertain the reliability of the CodeBLEU metric in assessing the generated test. Specifically, we expect that tests with higher CodeBLEU scores to have a higher likelihood of passing and vice-versa. Overall, the average score among passing and failing tests is 94% and 74%, respectively, suggesting that higher CodeBLEU scores are an indicator of passing tests. In particular, among the passing tests, 16 out of 24 achieved a perfect CodeBLEU score of 100%, indicating an exact match with the developer-repaired versions. Additionally, 22 tests scored above or equal to 80% on CodeBLEU. Conversely, among the 11 failed tests, only five tests achieved CodeBLEU scores above or equal to 80%. However, one passing test exhibited a notably lower CodeBLEU score of 36%, as illustrated in Figure 4.8, showing an alternative but correct repair. This particular case perfectly illustrates the limitations of using CodeBLEU scores.

To quantitatively model the relationship between CodeBLEU scores and the test passing probabilities, we train a Logistic Regression (LR) [42] model on the 35 executable tests, using their pass/fail labels, achieving an accuracy of 80% (correctly predicted test cases). The choice of LR is due to the small sample and the fact that we have one explanatory variable, CodeBLEU, that is expected to have a monotonic relationship with the test passing probability. The model's goodness of fit, measured by Chi-Square test [64], yields a p-value of 0.010, which indicates a statistically significant relationship between CodeBLEU scores and the probability for a test to pass, despite the small size of the sample. The fitted LR curve between CodeBLEU scores and the probability for a test to pass is shown in Figure 4.5, showing a sharp increase in passing probability beyond CodeBLEU scores of 50.

To estimate the passing test rate among the non-executable tests, we apply this trained

LR model to predict the outcomes of these tests. We calculate 95% confidence intervals for the predicted probabilities using standard errors derived via the delta method [7]. This method allows us to approximate the variance of the predicted probabilities by first calculating the standard errors on the logit scale and then transforming these to the probability scale. With this, we ensure that the confidence intervals account for the non-linear relationship between the logit and the predicted probability, giving a reliable measure of uncertainty around our predictions. According to our estimates, from the 181 test dataset, we obtain the following probability confidence intervals and passing test estimates:

- With No Label prompt: Confidence Interval: 0.44–0.50 — 80 (44%) tests to 91 (50%) tests are projected to pass.
- With Label prompt: Confidence Interval: 0.51–0.57 — 92 (51%) tests to 103 (57%) tests are projected to pass.

From the 131-test dataset (those that we can apply in-context learning on), we obtain the following results:

- With No Label prompt: Confidence Interval: 0.44–0.52 — 58 (44%) tests to 68 (52%) tests are projected to pass.
- With Label prompt: Confidence Interval: 0.50–0.58 — 66 (50%) tests to 76 (58%) tests are projected to pass.
- With in-context learning: Confidence Interval: 0.53–0.60 — 69 (53%) tests to 79 (60%) tests are projected to pass.

Tables 4.4 and 4.5 provide the passing test estimates in terms of upper and lower bounds across different prompt settings, for the 181 and 131 test datasets, respectively.

From the above results, we can conclude that providing labels with the prompt to GPT does indeed make a difference in terms of passing tests. Overall, a reasonable proportion of generated tests with labels are predicted to pass, ranging approximately from 50% to 60% when accounting for uncertainty. Furthermore, for the 11 GPT-generated tests that failed upon execution, we calculated the edit distance as a surrogate measure to assess the manual effort required for fixing. The results revealed that, on average, 16% of the tokens needed to be replaced to convert a failed test into the corresponding ground truth.Figure 4.6 depicts the trend between CodeBLEU scores and the percentage of token replacement. The scatter plot suggests that GPT-repaired tests with higher CodeBLEU scores tend to require fewer edits for repair, suggesting closer alignment with the developer-repaired versions. Lastly, to gain further insights into GPT's performance, we analyzed the fix categories of the 11 failed tests. Among these, six tests fell under the *Change Data Format* category. Of the remaining five, three were categorized under *Change Assertion* and two under *Change Data Structure*. The predominance of failed tests in the *Change Data Format* category can be attributed to the absence of in-context learning for this category, due to insufficient available instances as explained in Section 4.3.4. Our approach for this category relied solely

on prompts with label information, affecting the quality of generations. However, as visible in Table 4.3, for the other three fix categories, the results indicate notable improvements in GPT's generation quality with the implementation of in-context learning.

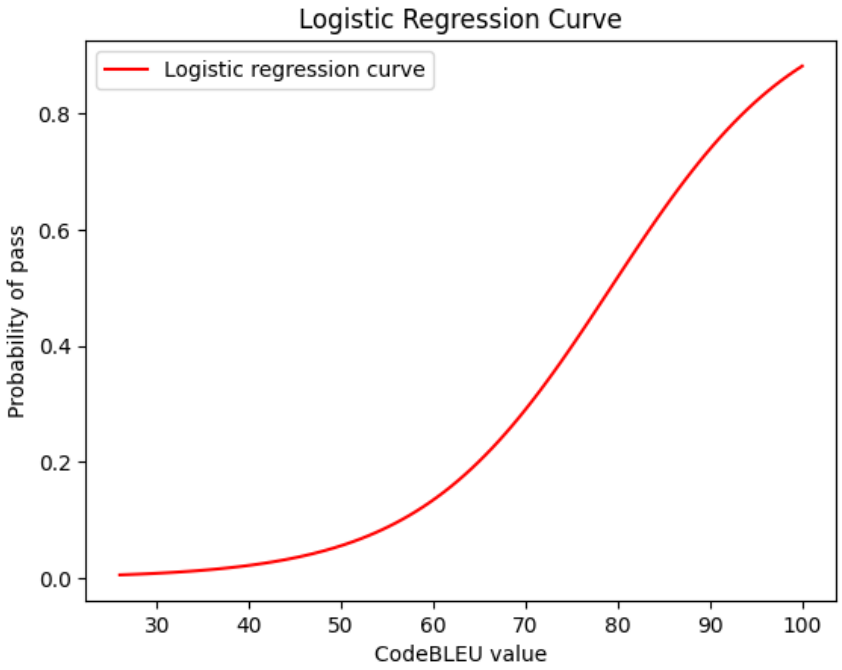



Figure 4.5: The Logistic Regression Curve showing a relation between CodeBLEU scores and probability of passing.

| Metrics | Prompts With Labels | | Prompts Without Labels | |
|---|---|---|---|---|
| | Mean | Median | Mean | Median |
| Corpus BLEU | 76 | 82 | 69 | 75 |
| Sentence BLEU | 77 | 82 | 70 | 76 |
| CodeBLEU | 81 | 85 | 75 | 81 |

Table 4.1: Evaluation of the GPT-3.5 generated flaky tests using BLEU Scores across all categories for 181 flaky tests.

## 4.4 Discussion

### 4.4.1 GPT-Generated Repair of Flaky Tests

To examine repaired flaky tests generated by GPT, we analyze three prompting methods: (a) a simple prompt without the fix category label, (b) the same prompt but with the

| Fix Categories | CodeBLEU Score | | | | Count |
|---|---|---|---|---|---|
| | Prompts With Labels | | Prompts Without Labels | | |
| | Mean | Median | Mean | Median | |
| Change Assertion | 80 | 85 | 76 | 79 | 64 |
| Change Condition | 87 | 90 | 84 | 89 | 41 |
| Change Data Structure | 87 | 81 | 72 | 82 | 41 |
| Misc | 76 | 83 | 75 | 82 | 22 |
| Change Data Format | 77 | 80 | 60 | 58 | 13 |
| Reorder Parameters | 86 | 86 | 77 | 83 | 10 |
| Reorder Data | 71 | 73 | 69 | 73 | 5 |
| Handle Exceptions | 66 | 73 | 59 | 60 | 7 |
| Reset Variable | 95 | 95 | 95 | 95 | 1 |

Table 4.2: Median and mean CodeBLEU score per fix category, with and without providing the fix category labels in the GPT prompt.

| Fix Categories | CodeBLEU Score | | | | | | |
|---|---|---|---|---|---|---|---|
| | Prompts Without Label | | Prompts With Label | | Prompts with In-Context Learning | | Count |
| | Mean | Median | Mean | Median | Mean | Median | |
| Change Assertion | 77 | 80 | 79 | 84 | 79 | 85 | 59 |
| Change Data Structure | 75 | 81 | 80 | 87 | 86 | 94 | 36 |
| Change Condition | 85 | 88 | 88 | 90 | 92 | 97 | 36 |

Table 4.3: Comparison of median and mean CodeBLEU scores for different fix categories using fix category labels and in-context learning.

| Prompts | Lower Bound | | Upper Bound | |
|---|---|---|---|---|
| | Passing Tests # | Passing % | Passing Tests # | Passing % |
| With No Label | 80 | 44% | 91 | 50% |
| With Label | 92 | 51% | 103 | 57% |

Table 4.4: Passing Estimates from the 181 test dataset for the GPT-repaired tests with and without providing the fix category labels in the GPT prompt.

| Prompts | Lower Bound | | Upper Bound | |
|---|---|---|---|---|
| | Passing Tests # | Passing % | Passing Tests # | Passing % |
| With No Label | 58 | 44% | 68 | 52% |
| With Label | 66 | 50% | 76 | 58% |
| In-Context Learning | 69 | 53% | 79 | 60% |

Table 4.5: Passing Estimates for the GPT-repaired tests using fix category labels and in-context learning (for the 131 tests).

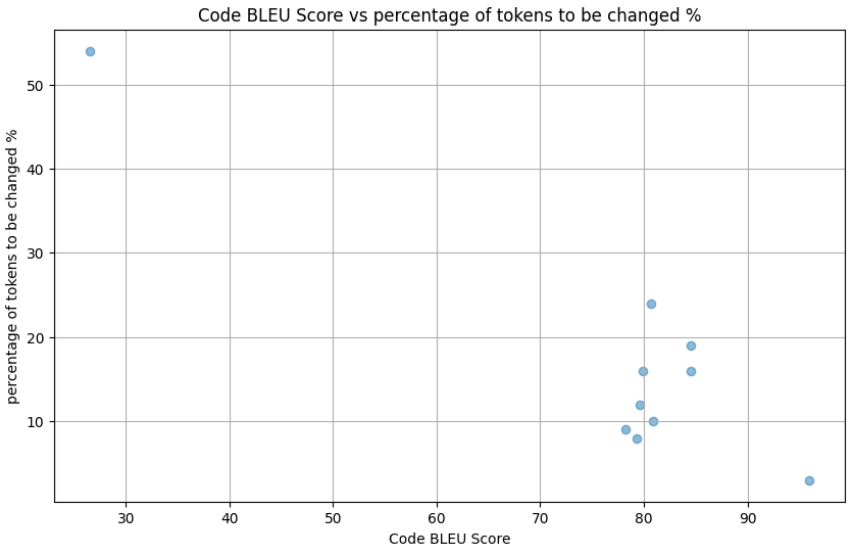



Figure 4.6: Comparison of CodeBLEU Score with the percentage of tokens changed in failed GPT-generated tests

label, and (c) adding extra context (sample fixes) to the prompt, i.e., in-context learning (described in Section 4.3.4).

Figure 4.13 depicts a flaky test and its fix by the original developer. Both UniXcoder and CodeBERT classified this as a *Change Data Structure* fix, replacing HashMap with LinkedHashMap to maintain the elements' order. Figure 4.14 shows the results based on prompting methods (a) without the *Change Data Structure* label and (b) with that label.

When the model was provided with the fix category label, the output matched the developer's repair (Figure 4.13 – right), achieving a 100% CodeBLEU Score. However, without the label indicating which statements to fix, the model suggested fixes on the wrong statements unrelated to the issue.

Now let's look at another example from the *Change Condition* category, showcased in Figure 4.15. Here, the repair modified *.containsExactly()* to *.containsExactlyInAnyOrder()* to fix flakiness. Figure 4.16 shows the repaired tests with and without the fix category label. It also exemplifies a repair where we provided guidance within the prompt for fixing flaky tests caused by change conditions.

In all generated instances, the generated fix is not an exact match to the original developer's repair. In the first two cases, *.containsExactly()* was replaced with *.containsAnyof()*, which is similar to the ground truth, but does not match exactly. The label-free model wrongly removed the exception handling. However, with the label, the model preserved the structure and attempted to fix the problematic statements.

In the third case, in-context learning, the model correctly replaced *.containsExactly()* with *.containsExactlyInAnyOrder()* based on what model learned from the examples we passed in the prompt. However, it also incorrectly removed proper exception handling, similar to the first instance. Based on our observation, it seems that GPT's accuracy in

generating fixes depends on whether the prompt contains the fix category and the number of examples. Future work could improve this accuracy by augmenting the training data and enhancing prompt quality.

## 4.5 Threats to Validity

### 4.5.1 Construct Threats

Furthermore, the accuracy of the results that we obtained for our predicted fix category labels are very high for most of the fix categories, as shown in Table 3.3. Given this, we ensured that correct predicted labels are used in different prompts to LLM in this RQ. Regarding the evaluation metrics in this RQ, we used the CodeBLEU score, a common metric in text generation tasks. However, we did not execute these automated generated tests to verify their functionality. Due to dependencies in open-source projects, executing all these tests is time and computation-intensive. As a result, we have not confirmed whether the tests with higher CodeBLEU scores will pass or not, or whether they are even executable. However, improvements in CodeBLEU are unlikely to yield worse test cases, in terms of functionality or execution. Nevertheless, the inadequacy of the current metrics is a known challenge [23] in this field and future research is needed to reduce such threats.

### 4.5.2 Internal Threats

Internal threats to validity are about to make sure the cause-effect relation identified in the study is really there and there is no other explanation (confounding factors). In our study, the claim is that *our predicted fix categories will help fix flakiness*. The main confounding factor is about the way we prompt the GPT-3.5 model, when asking to fix a flaky test. Better results might be obtained in the future with additional guidance and more examples in the prompt. Similarly, using GPT-4 would probably lead to even better results but, as discussed in Section 4.2.1, this would raise, on our data set, possible data leakage issues. Additionally, it would be insightful to analyze how often GPT generates a correct fix when provided with an incorrectly predicted fix category. However, as explained in Section 4.3.3, we were limited to executing only 35 tests. Within our sample of 35 executable test cases, we encountered only one instance where the fix category was incorrectly predicted, albeit partially correct as one of the two categories was accurately predicted. Therefore, conducting this analysis reliably is not possible with our current data.

### 4.5.3 Conclusion Threats

To reduce potential conclusion validity threats, we ran Wilcoxon signed-rank test when comparing different prompt types in RQ6. However, there is still a potential conclusion threat in RQ6 results with respect to the inherent randomness of GPT models. Indeed, using the same model and the same prompt can result in different outputs when using

higher temperatures. In general, the field of prompt engineering, which also includes the optimization of outputs with several rounds of prompting, is a new field and there are limited precise and reliable guidelines in the literature. Future work will investigate future prompting strategies as they get reported.

### 4.5.4 External Threats

We have only evaluated our repair approach on the tests written in Java and not other programming languages but it can be adapted to other languages as GPT models are pre-trained on multiple programming languages.

## 4.6 Related Work

In this section, we provide an overview of the most related work to predicting flaky test fix categories and their repair. There are two areas of research that we deliberately exclude, since they are less related to our study. The first category that we do not include is generic program repair studies [32]. Although, fixing test code can be seen as a similar task to program repair, there are fundamental differences between the two areas which are mainly due to the patterns of fixes. A generic code repair can come in many flavors and patterns and thus predicting a category of fix is a daunting task. In addition, there are far more data available for generic program repair (e.g., literally any code fix in GitHub) than for fixing test flakiness. Therefore, we do not discuss individual papers that study generic program repair. The second category that we exclude here are studies that analyze test smells and their patterns [31]. Although these works are more relevant since they are about issues in the test code, given that they are again generic and not related to flakiness, we exclude them.

The first set of relevant papers to our study are those that specifically address categories or taxonomies of fixes or fix patterns. Although they are not about flaky test fixes or not even about test cases, these studies have nevertheless a similar problem to solve: categorizing code fixes or causes and predicting them. Ni et al [108] proposed an approach to classify bug fix categories using a Tree-based Convolutional Neural Network (TBCNN). To do so, they used the *diff* of the source code before and after the fix at the Abstract Syntax Tree (AST) level to construct fix trees as input for TBCNN. In contrast, our objective is to develop a classifier for fix categories that only use flaky test code to make predictions, without relying on the repaired versions or production code.

Predicting the cause of flakiness is another category that is relevant to our work. Akli et al. proposed a novel approach [4] for classifying flaky tests based on their root cause category using CodeBERT and FSL. They manually labeled 343 flaky tests with 10 different categories of flaky test causes suggested by Luo et al [60] and Eck et al [21]. However, they reported the prediction results for only four categories of causes, namely Async wait, Concurrency, Time, and Unordered collection, as they lacked sufficient positive examples to train an ML-based classifier for the other categories. They achieved the highest prediction

results for the Unordered collection cause, with a precision of 85% and a recall of 80%. In a recent survey on flaky tests [51,85], Parry et al. mapped three different causes of flakiness with their potential repairs. For instance, they suggested adding or modifying assertions in the test code to fix the flakiness caused by concurrency. Predicting fix categories provides testers with a specific direction for modifying the code. Knowing that concurrency is causing flakiness in a given test is not as useful to testers as they do not know which part of the code is causing the issue. However, knowing that changing assertions can eliminate flakiness gives them more practical guidance. Other researchers reported studies [51,128] to identify flakiness causes, impacts, and existing strategies to fix flakiness. However, none of them attempted to categorize the tests based on their fixes. Research on repairing flakiness and categorizing fix categories is still in its early stages, with much of the work focused on fixing order-dependent flaky tests. For example, iPFlakies [113] and iFixFlakies [103] propose methods for addressing order-dependent flakiness, while Flex [20] automates the repair of flaky tests caused by algorithmic randomness in ML algorithms. Parry et al. [87] proposed a study in which different categories of flaky test fixes were defined and 75 flaky commits were categorized based on these categories. However, the categorization was performed through manual inspection of the commits, code diffs, developer comments, and other linked issues outside the scope of the test code. In another study of flaky tests in the Mozilla database [21] tests were categorized based on their origin (i.e., the root cause of the flakiness, whether in the source or test code) and developers' efforts to fix them. The need for automated fixing of flaky tests and conducting quick small repairs was strongly suggested in a recent survey on supporting developers and testers in addressing test flakiness [34]. Our study aims to address these gaps by developing an automated solution to categorize flaky tests based on well-defined fix categories, relying exclusively on test code, providing practical guidance for both manual and automated repair. We also provide a way to automatically label flaky tests with selected, common fix categories to train context-specific prediction models.

Although we do not review the generic automated program repair literature, as explained, we do however discuss related work where LLMs are used for automatic program repair, since it is very relevant to RQ6 from a technical standpoint. Recent advancements in LLMs, exemplified by ChatGPT [30], Copilot [76], and similar models tailored for code, have significantly advanced this field. These models exhibit adeptness in code generation and are generating considerable interest among researchers striving to enhance automatic program repair methodologies.

Ribeiro et al. [99] employed GPT-3 to automatically rectify bugs in 1318 buggy OCaml programs, achieving a repair rate of 39%. Julian et al. [95], using OpenAI's Codex model, repaired bugs in Java and Python programs, noting a higher repair rate for Python. Their utilization of prompts and hints improved Codex's bug identification, yielding slightly enhanced results in repairing Java programs. In another recent study by Chunqiu et al. [120], different LLMs like Codex, InCoder, CodeT5 and GPT-2 are investigated for program repair in Java, Python, and C programs.

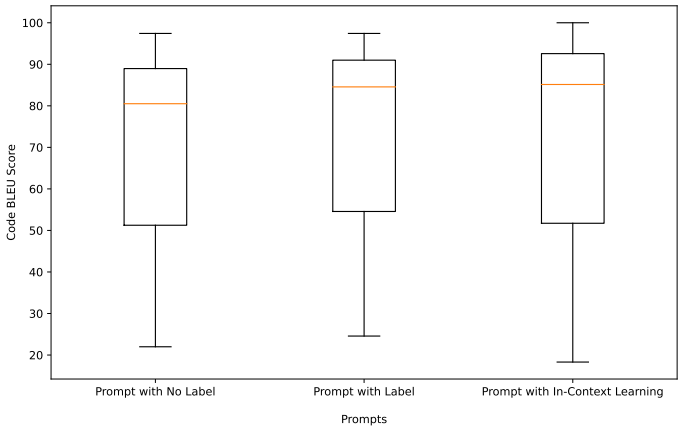
Additionally, Mashadi et al. [62] and Xia et al. [121] utilized CodeBERT for automated program repair on datasets like ManySStuBs4J and Defects4J, respectively, achieving state-of-the-art results in bug repair for smaller code segments due to CodeBERT's

input limitation of up to 512 tokens.

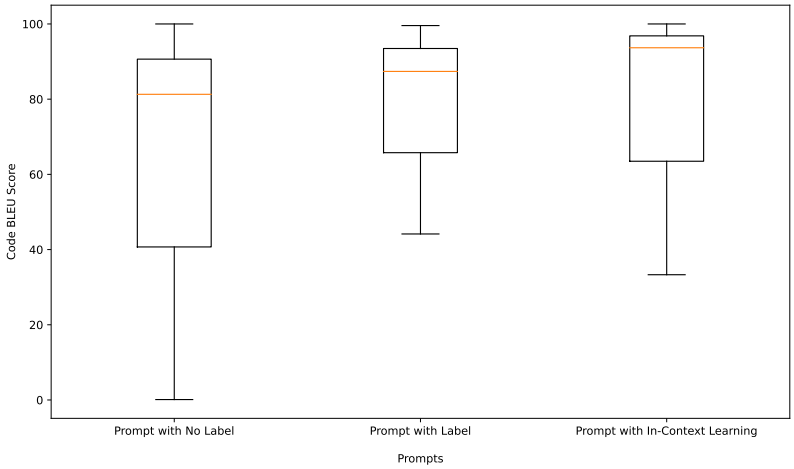
However, while existing research has focused on rectifying bugs in source code, little attention has been directed towards addressing flakiness within test code itself. Our study marks the first attempt to fully automate flaky test repairs using GPT, complemented by providing fix category information to pinpoint flakiness related statements in the code and enhance the repair capability of the GPT model.

## 4.7 Conclusion

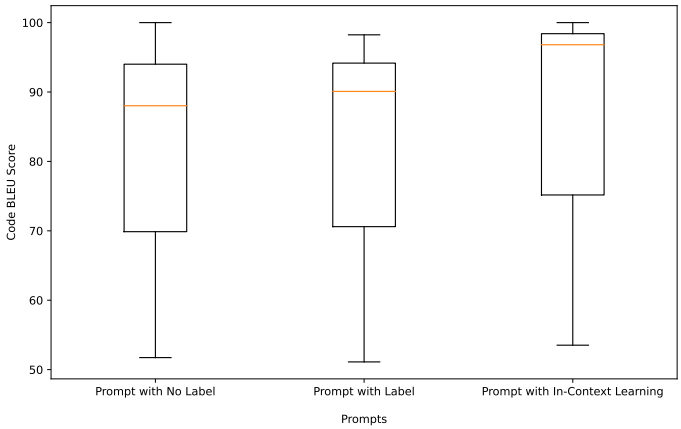
In this chapter, we use the fix categories discussed in chapter 3 to generate fully automated repairs of the flaky tests with LLMs. We showed that the category information, complemented with in-context learning can greatly help a LLM, namely GPT-3.5 in our case, better fix the flakiness in test code. Moreover, we executed a sample of GPT-repaired flaky tests and estimated that a large percentage (roughly between 51% and 83%) of these tests can be expected to pass. For the generated tests that fail, an average of 16% of the code needs further modification for them to pass.




(a) Change Assertion




(b) Change Data Structure




(c) Change Condition

Figure 4.7: Distribution of CodeBLEU scores with respect to three prompts.

```
@Test
public void queryByEmptyKeySubObject() {
    JSONObject json = new JSONObject("{\"\":\"empty key of an object with an empty key\",\"subKey\":\"Some" +
                " other value\"}");
    JSONObject obj = (JSONObject) query("/obj/");
    assertTrue(json.similar(obj));
}
```

```
@Test
public void queryByEmptyKeySubObject() {
    JSONObject expectedJsonObject = new JSONObject();

    expectedJsonObject.put("", "empty key of an object with an empty key");
    expectedJsonObject.put("subKey", "Some other value");
    assertEquals(expectedJsonObject.toString(), query("/obj/").toString());
}

```

Figure 4.8: Comparison between a GPT-repaired flaky test (right) and its original developer repair (left), demonstrating a case where the GPT-repaired test passed upon execution despite displaying a very low CodeBLEU score of 36%.

```
@Test public void testFetchToAttributesWithStringValues(){
  final Map<String,String> cells=new HashMap<>();
  cells.put("cq1","val1");
  cells.put("cq2","val2");
  final long ts1=123456789;
  hBaseClientService.addResult("row1",cells,ts1);
  runner.setProperty(FetchHBaseRow.AUTHORIZATIONS,"");
  runner.setProperty(FetchHBaseRow.TABLE_NAME,"table1");
  runner.setProperty(FetchHBaseRow.ROW_ID,"row1");
  runner.setProperty(FetchHBaseRow.DESTINATION,FetchHBaseRow.DESTINATION_ATTRIBUTES);
  runner.enqueue("trigger flow file");
  runner.run();
  runner.assertTransferCount(FetchHBaseRow.REL_FAILURE,0);
  runner.assertTransferCount(FetchHBaseRow.REL_SUCCESS,1);
  runner.assertTransferCount(FetchHBaseRow.REL_NOT_FOUND,0);
  final MockFlowFile flowFile=runner.getFlowFilesForRelationship(FetchHBaseRow.REL_SUCCESS).get(0);
  flowFile.assertAttributeEquals(FetchHBaseRow.HBASE_ROW_ATTR,"{\"row\":\"row1\", \"cells\": [" + "{\"fam\":\"nifi\",\"qual\":\"cq1\",\"val\":\"val1\",\"ts\":" + ts1 + "}, "+ "{\"fam\":\"nifi\",\"qual\":\"cq2\",\"val\":\"val2\",\"ts\":"+ ts1+ "}]}");
  Assert.assertEquals(1,hBaseClientService.getNumScans());
 }
```

Figure 4.9: Flaky test example with correct mapping to Change Data Structure fix category as HashMap here should be replaced with LinkedHashMap to remove flakiness.

```
@Test public void testAllowNullSchema(){
  JSONSchema<Foo> jsonSchema=JSONSchema.of(SchemaDefinition.<Foo>builder().withPojo(Foo.class).build());
  Assert.assertEquals(jsonSchema.getSchemaInfo().getType(),SchemaType.JSON);
  Schema.Parser parser=new Schema.Parser();

  String schemaJson=new String(jsonSchema.getSchemaInfo().getSchema());
  Assert.assertEquals(schemaJson,SCHEMA_JSON_ALLOW_NULL);
  Schema schema=parser.parse(schemaJson);
  for (  String fieldName : FOO_FIELDS) {
    Schema.Field field=schema.getField(fieldName);
    Assert.assertNotNull(field);
    if (field.name().equals("field4")) {
      Assert.assertNotNull(field.schema().getTypes().get(1).getField("field1"));
    }
    if (field.name().equals("fieldUnableNull")) {
      Assert.assertNotNull(field.schema().getType());
    }
  }
}

```

(a) Flaky Test

```
@Test public void testAllowNullSchema() throws JSONException {
  JSONSchema<Foo> jsonSchema=JSONSchema.of(SchemaDefinition.<Foo>builder().withPojo(Foo.class).build());
  Assert.assertEquals(jsonSchema.getSchemaInfo().getType(),SchemaType.JSON);
  Schema.Parser parser=new Schema.Parser();
  parser.setValidateDefaults(false);
  String schemaJson=new String(jsonSchema.getSchemaInfo().getSchema());
  assertJSONEqual(schemaJson,SCHEMA_JSON_ALLOW_NULL);
  Schema schema=parser.parse(schemaJson);
  for (  String fieldName : FOO_FIELDS) {
    Schema.Field field=schema.getField(fieldName);
    Assert.assertNotNull(field);
    if (field.name().equals("field4")) {
      Assert.assertNotNull(field.schema().getTypes().get(1).getField("field1"));
    }
    if (field.name().equals("fieldUnableNull")) {
      Assert.assertNotNull(field.schema().getType());
    }
  }
}

```

(b) Fixed Flaky Test

Figure 4.10: Example of Flaky test (Top) and its developer repair (Bottom) where flakiness is removed by changing assertion and adding exception handling. Our prediction model correctly classifies this flaky test under the Handle Exception and Change Assertion fix category.

```
@Test public void manyParameters() throws Exception {
  assertThat(testResult(ManyParameters.class),isSuccessful());
  assertEquals(16,ManyParameters.iterations);
  assertEquals(asList(-1,-2,-4),ManyParameters.firstTestCases.subList(0,3));
  assertEquals(asList(-1,-2,-4),ManyParameters.firstTestCases.subList(4,7));
  assertEquals(asList(-1,-2,-4),ManyParameters.firstTestCases.subList(8,11));
  assertEquals(asList(-1,-2,-4),ManyParameters.firstTestCases.subList(12,15));
  assertEquals(asList('r','r','r','r','y','y','y','y'),ManyParameters.secondTestCases.subList(0,8));
}
```

(a) Flaky test

```
@Test public void manyParameters(){
  assertThat(testResult(ManyParameters.class),isSuccessful());
  assertEquals(16,ManyParameters.iterations);
  assertEquals(asList(-4,-2,-1),ManyParameters.firstTestCases.subList(0,3));
  assertEquals(asList(-4,-2,-1),ManyParameters.firstTestCases.subList(4,7));
  assertEquals(asList(-4,-2,-1),ManyParameters.firstTestCases.subList(8,11));
  assertEquals(asList(-4,-2,-1),ManyParameters.firstTestCases.subList(12,15));
  assertEquals(asList('r','r','r','r','y','y','y','y'),ManyParameters.secondTestCases.subList(0,8));
}
```

(b) Fixed flaky test

Figure 4.11: Example of Flaky test (Top) and its developer repair (Bottom) where our prediction model is unable to correctly classify this flaky test under the Handle Exception fix category.

```
@Test public void testGenerateSearchWeightString(){
  Map<String,Integer> fields=new HashMap<>();
  fields.put("one",10);
  fields.put("two",1);
  fields.put("three",15);
  String generatedString=AtlasJanusGraphIndexClient.generateSearchWeightString(fields);
  Assert.assertEquals(generatedString," one^10 two^1 three^15");
 }
```

Figure 4.12: Example of a flaky test where the model incorrectly classifies it in the Change Assertion fix category.

```
@Test
    public void testParamsMapToString() {
        Map<String, String[]> params = new HashMap<>();
        params.put("a", new String[]{"1", "2", "3"});
        params.put("b", new String[]{"8", "7", "6"});
        String result = UrlUtils.paramsMapToString(params);
        Assert.assertEquals(result, "&a=1,2,3&b=8,7,6");
    }
```

```
@Test
    public void testParamsMapToString() {
        Map<String, String[]> params = new LinkedHashMap<>();
        params.put("a", new String[]{"1", "2", "3"});
        params.put("b", new String[]{"8", "7", "6"});
        String result = UrlUtils.paramsMapToString(params);
        Assert.assertEquals(result, "&a=1,2,3&b=8,7,6");
    }
```

Figure 4.13: Example of a flaky test (left) and its original developer repair (right). Here flakiness is removed by changing the data structure i.e. replacing HashMap with LinkedHashMap.

```
@Test
public void testParamsMapToString() {
    Map<String, String[]> params = new HashMap<>();
    params.put("a", new String[]{"1", "2", "3"});
    params.put("b", new String[]{"8", "7", "6"});
    String result = UrlUtils.paramsMapToString(params);
    String expectedResult = "&a=1,2,3&b=8,7,6";
    Assert.assertTrue(result.contains("&a=1,2,3"));
    Assert.assertTrue(result.contains("&b=8,7,6"));
}
```

```
@Test
public void testParamsMapToString() {
    Map<String, String[]> params = new LinkedHashMap<>();
    params.put("a", new String[]{"1", "2", "3"});
    params.put("b", new String[]{"8", "7", "6"});
    String result = UrlUtils.paramsMapToString(params);
    Assert.assertEquals(result, "&a=1,2,3&b=8,7,6");
}

```

Figure 4.14: Comparison between GPT-generated repaired flaky tests without a fix category label prompting (left) and with a fix category label provided(right), illustrating GPT's accurate generation of the repaired flaky test when given the fix category label in the prompt.

```
@Test
    public void testBuildTagSome() {
        try {
            SubscriptionData subscriptionData = FilterAPI.build(
                    "TOPIC", "A || B", ExpressionType.TAG
            );
            assertThat(subscriptionData).isNotNull();
            assertThat(subscriptionData.getTopic()).isEqualTo("TOPIC");
            assertThat(subscriptionData.getSubString()).isEqualTo("A || B");
            assertThat(ExpressionType.isTagType(subscriptionData.getExpressionType())).isTrue();

assertThat(subscriptionData.getTagsSet()).isNotNull();
            assertThat(subscriptionData.getTagsSet()).containsExactly("A", "B");
        } catch (Exception e) {
            e.printStackTrace();
            assertThat(Boolean.FALSE).isTrue();
        }
    }
```

```
@Test
    public void testBuildTagSome() {
        try {
            SubscriptionData subscriptionData = FilterAPI.build(
                    "TOPIC", "A || B", ExpressionType.TAG
            );
            assertThat(subscriptionData).isNotNull();
            assertThat(subscriptionData.getTopic()).isEqualTo("TOPIC");
            assertThat(subscriptionData.getSubString()).isEqualTo("A || B");
            assertThat(ExpressionType.isTagType(subscriptionData.getExpressionType())).isTrue();

assertThat(subscriptionData.getTagsSet()).isNotNull();
            assertThat(subscriptionData.getTagsSet()).containsExactlyInAnyOrder("A", "B");
        } catch (Exception e) {
            e.printStackTrace();
            assertThat(Boolean.FALSE).isTrue();
        }
    }
```

Figure 4.15: Example of a flaky test (left) and its original developer repair (right). Here flakiness is removed by changing the condition, i.e., replacing .containsExactly() with .containsExactlyInAnyOrder().

```
@Test
public void testBuildTagSome() {
    try {
        SubscriptionData subscriptionData = FilterAPI.build(
                "TOPIC", "A || B", ExpressionType.TAG
        );
        assertThat(subscriptionData).isNotNull();
        assertThat(subscriptionData.getTopic()).isEqualTo("TOPIC");
        assertThat(subscriptionData.getSubString()).isEqualTo("A || B");
        assertThat(ExpressionType.isTagType(subscriptionData.getExpressionType
())).isTrue();

        assertThat(subscriptionData.getTagsSet()).isNotNull();
        assertThat(subscriptionData.getTagsSet()).containsAnyOf("A", "B");
    } catch (Exception e) {
        e.printStackTrace();
        assertThat(false).isTrue();
    }
}

```

(a) Repaired flaky test with fix category label.

```
@Test
public void testBuildTagSome() {
    SubscriptionData subscriptionData = FilterAPI.build(
            "TOPIC", "A || B", ExpressionType.TAG
    );
    assertThat(subscriptionData).isNotNull();
    assertThat(subscriptionData.getTopic()).isEqualTo("TOPIC");
    assertThat(subscriptionData.getSubString()).isEqualTo("A || B");
    assertThat(ExpressionType.isTagType(subscriptionData.getExpressionType())).
isTrue();
    assertThat(subscriptionData.getTagsSet()).isNotNull();
    assertThat(subscriptionData.getTagsSet()).containsAnyOf("A", "B");
}
```

(b) Repaired flaky test without fix category label.

```
@Test
public void testBuildTagSome() {
    SubscriptionData subscriptionData = FilterAPI.build(
            "TOPIC", "A || B", ExpressionType.TAG
    );
    assertThat(subscriptionData).isNotNull();
    assertThat(subscriptionData.getTopic()).isEqualTo("TOPIC");
    assertThat(subscriptionData.getSubString()).isEqualTo("A || B");
    assertThat(ExpressionType.isTagType(subscriptionData.getExpressionType())).
isTrue();

    assertThat(subscriptionData.getTagsSet()).isNotNull();
    assertThat(subscriptionData.getTagsSet()).containsExactlyInAnyOrder("A",
"B");
}

```

(c) Repaired flaky test with in-context learning.

Figure 4.16: Effect of different prompts on GPT's repaired flaky test generation: with fix label (Top), without fix label (middle), and with in-context learning (bottom).

# Chapter 5

# Conclusion

In this section, a summary of contributions is provided along with the practical implications of this work. Moreover, we also discuss this thesis' limitations. Finally, future work items are identified.

## 5.1 Summary of Contributions

In summary, this thesis provides the following contributions:

1. *Flakify*, a generic, black-box, language model-based flaky test case predictor, which does not require rerunning test cases. It predicts flaky test cases on the basis of test code without requiring the definition of features. This is the first time a language model has been used to address flaky tests. Our contributions related to Flakify include:

   (a) Development of ML-based classifier that predicts flaky test cases based on the test code without requiring the definition of features.

   (b) A Java-based implementation of an Abstract Syntax Tree (AST)-based technique for statically detecting and only retaining statements that match eight test smells in the test code, thus enhancing the application of LMs.

   (c) Running a large scale experiment, predicting flaky tests on two distinct datasets: FlakeFlagger containing 21,661 test cases collected from 23 Java projects, and the International Dataset of Flaky Tests (IDoFT) containing 3,862 test cases collected from 312 Java projects.

2. *FlakyFix*, a novel black box language model-based framework for categorizing flaky tests on the basis of the fixes required and then automatically repairing them. Our contributions related to FlakyFix are:

   (a) Definition of a categorization for flaky test fixes. This is done to provide guidance, either to a human or a LLM, for fixing tests.

(b) Development of a set of heuristics (search rules) and accompanying Python based open-source scripts to automatically label flaky tests based on their fixes.

(c) Creation of a public dataset of labeled flaky tests for training, using our automated script.

(d) Definition of a black-box framework for predicting flaky test fix categories using code models (LM that are pre-trained on code) and few-shot learning (FSL).

(e) Quantitative and qualitative analysis of two SOTA small code models, with and without FSL, to compare alternative prediction models.

(f) Automatic generation of fixes for flaky tests using a LLM (GPT 3.5 Turbo), with and without the predicted fix categories, to analyze how these labels enhance the generated fixes.

(g) Based on the execution of a sample of GPT-generated tests, we relied on CodeBLEU metrics and statistical methods, including bootstrapping and logistic regression, to estimate the pass rate for all generated repaired tests, including those that could not be executed. We also assessed the magnitude of the modifications needed to manually fix tests that failed.

## 5.2 Practical Implications

Our research aims to significantly improve industrial practices by addressing test flakiness. We seek to reduce developers' efforts in detecting flaky tests, eliminating the need to rerun tests multiple times, especially in large-scale systems. Additionally, we propose methods to categorize tests based on fix types and to automate full or partial repairs, enhancing the reliability of the software testing process.The data sets and codes used in Flakify and FlakyFix are publicly available in our replication packages [25, 26].

### 5.2.1 Flakify in Practice

Flakify can be deployed in Continuous Integration (CI) environments to help detect flaky test cases. One could argue that the CI build history can be used as reference to conclude whether a test case is flaky or not. However, regular test case executions across builds may not entirely solve the problem, since differences in test case verdicts, i.e., pass or fail, can be due to differences in builds rather than flakiness. Therefore, test engineers can use the prediction results obtained from Flakify to fix test cases that are predicted as flaky, e.g., by eliminating the presence of test smells, or otherwise rerun them a larger number of times, using the same code version, to verify whether a test case is actually flaky or not. More specifically, Flakify helps test engineers focus their attention on a small subset of test cases that are most likely to be flaky in a CI build. As our results show, Flakify significantly reduces the cost of debugging test and production code, both in terms of human effort and execution time. This makes Flakify an important strategy in practice to achieve scalability, especially when applied to large test suites. Moreover, the test smell detection capability

of Flakify helps to inform test engineers about possible causes of flakiness that need to be addressed.

### 5.2.2 FlakyFix in Practice

FlakyFix can help repair flaky tests both automatically or with some human intervention. When a test is flagged as problematic (either by a person or other tools), there are two options to fix flakiness: a manual fix by the tester, or running an automated program repair tool. In either case, our approach is helpful. If the repair is manual, the predicted fix categories from our approach are given to the testers as suggestions so that they can more quickly fix the bug in the test code. Alternatively, if a LLM such as GPT is used to fix the test, the category labels are helpful for prompting, as shown in the previous section. However, as discussed in Section 4.3.7, not all GPT-repaired tests are completely correct and will pass upon execution. This is primarily reflected in the CodeBLEU score and the execution results obtained after running a subset of generated tests. We estimate, however, that a large majority of GPT-repaired test cases should pass, their percentage depending on the specific prompt being used. A failed generated test will require some manual repair for it to pass. Our analysis using edit distance reveals that, on average, 16% of the tokens need to be changed in a failing test for it to exactly match the passing, original-developer repaired test. Thus, completely fixing flaky test cases does entail some degree of human effort for a minority of repaired test cases and is not entirely free. However, with newer GPT versions and better prompting, we can hope for increasingly reliable fixes and passing test cases in the future.

## 5.3 Limitations

The limitations of the results can be summarized as follows:

- **Dataset:** We evaluated both Flakify and FlakyFix using the IDoFT and FlakeFlagger datasets, which contain Java-based test cases. Although these frameworks were not tested on other programming languages, we utilized LLMs such as CodeBERT, UniXcoder, and GPT, which are trained in multiple languages. Thus, we believe our techniques are adaptable. However, the IDoFT dataset used is relatively small. To generalize our findings, we need to evaluate larger datasets across different programming languages.
- **Test code input length:** On average, 12% of the test cases from the IDoFT dataset exceed CodeBERT's 512-token limit, and 4% exceed UniXcoder's 1024-token limit, potentially causing information loss and affecting performance. Exploring models with longer token limits could help, though it may reduce efficiency.
- **Better engineering tool for usability:** Flakify and FlakyFix should be integrated into a single engineering tool to enhance usability as an API plugin or browser extension. Since Flakify detects flaky tests and FlakyFix repairs them, combining them

into one tool would streamline the process for developers, which is currently lacking in our approach.

- **Use of open source LLMs:** In this research, we used the GPT model to generate flaky test repairs. However, GPT requires paid API calls. To scale our technique for a higher number of flaky tests, as typically seen in industrial settings, we need to explore less expensive, open-source models that offer comparable performance.

## 5.4 Future Work

In this research, we focused on black-box solutions to eliminate flaky tests, specifically by detecting and fixing flakiness using only the test code. However, a significant proportion of flaky tests can arise from issues in the production code, which cannot be addressed by black-box models. Therefore, future work should focus on developing lightweight and scalable approaches to tackle these root causes of flakiness.

Additionally, existing public datasets are no ideal for evaluating flaky test case prediction approaches, as the ratio of flaky test cases is typically low. The detection of flaky test cases in these datasets often relies on rerunning tests multiple times while monitoring their behavior, a technique that may not always yield accurate results. Furthermore, many open-source projects now utilize Continuous Integration (CI), which provides extensive test execution histories. Given the frequency of test executions in CI environments and the high workload on CI servers, test cases may exhibit additional flakiness behaviors that are not revealed when running tests on dedicated machines.

To address these limitations, we plan to build a larger dataset of flaky test cases specifically targeting CI contexts. We also intend to expand our flaky dataset with more data and enhance our heuristics for automatically labeling flaky tests across different programming languages. Future studies will involve replicating our research with other LLMs using an updated dataset.

Moreover, we recognize that other types of flaky tests, such as order-dependent and asynchronous flaky tests, require attention. In the future, we should extend our work to address these issues as well.

# References


[1] Identifying and analyzing flaky tests in Maven and Gradle builds. `https://gradle.com/blog/flaky-tests`. Accessed: 2021-11-01.

[2] Mohammad Mahdi Abdollahpour, Mehrdad Ashtiani, and Fatemeh Bakhshi. Automatic software code repair using deep learning techniques. *Software Quality Journal*, pages 1–30, 2023.

[3] Abien Fred Agarap. Deep learning using rectified linear units (ReLU). *arXiv preprint arXiv:1803.08375*, 2018.

[4] Amal Akli, Guillaume Haben, Sarra Habchi, Mike Papadakis, and Yves Le Traon. Flakycat: Predicting flaky tests categories using few-shot learning. In *2023 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 140–151. IEEE, 2023.

[5] Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D Newman, Abdullatif Ghallab, and Stephanie Ludi. Test smell detection tools: A systematic mapping study. *Evaluation and Assessment in Software Engineering*, pages 170–180, 2021.

[6] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. Flakeflagger: Predicting flakiness without rerunning tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1572–1584. IEEE, 2021.

[7] Susan F Assmann, David W Hosmer, Stanley Lemeshow, and Kenneth A Mundt. Confidence intervals for measures of interaction. *Epidemiology*, pages 286–290, 1996.

[8] Nguyen Bach and Sameer Badaskar. A review of relation extraction. *Literature review for Language and Statistics II*, 2:1–15, 2007.

[9] Thomas Bach, Artur Andrzejak, and Ralf Pannemans. Coverage-based reduction of test execution time: Lessons from a very large industrial project. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 3–12. IEEE, 2017.

[10] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. Deflaker: Automatically detecting flaky tests. In *Proceedings of the 40th international conference on software engineering*, pages 433–444, 2018.

[11] Paula Branco, Luís Torgo, and Rita P Ribeiro. A survey of predictive modeling on imbalanced domains. *ACM Computing Surveys (CSUR)*, 49(2):1–50, 2016.

[12] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[13] Bruno Camara, Marco Silva, Andre Endo, and Silvia Vergilio. On the use of test smells for prediction of flaky tests. In *Brazilian Symposium on Systematic and Automated Software Testing*, pages 46–54, 2021.

[14] Bruno Henrique Pachulski Camara, Marco Aurélio Graciotto Silva, André Takeshi Endo, and Silvia Regina Vergilio. What is the Vocabulary of Flaky Tests? An Extended Replication. In *2021 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC) (ICPC)*, pages 444–454, 2021.

[15] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[16] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*, 2020.

[17] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of machine learning research*, 12(ARTICLE):2493–2537, 2011.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[19] Yihong Dong, Jiazheng Ding, Xue Jiang, Zhuo Li, Ge Li, and Zhi Jin. Codescore: Evaluating code generation by learning code execution. *arXiv preprint arXiv:2301.09043*, 2023.

[20] Saikat Dutta, August Shi, and Sasa Misailovic. Flex: fixing flaky tests in machine learning projects by updating assertion bounds. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 603–614, 2021.

[21] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. Understanding flaky tests: The developer's perspective. In *Proceedings of the 2019 27th ACM*

*Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 830–840, 2019.

[22] Salma El Anigri, Mohammed Majid Himmi, and Abdelhak Mahmoudi. How BERT's dropout fine-tuning affects text classification? In *International Conference on Business Intelligence*, pages 130–139. Springer, 2021.

[23] Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. Out of the bleu: how should we assess quality of the code generation models? *Journal of Systems and Software*, 203:111741, 2023.

[24] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533*, 2023.

[25] Sakina Fatima. Flakify: A Black-Box, Language Model-based Predictor for Flaky Tests – Replication Package. `https://doi.org/10.5281/zenodo.6994692`.

[26] Sakina Fatima. FlakyFix:Using Large Language Models for Predicting Flaky Test Fix Categories and Test Code Repair – Replication Package. `https://figshare.com/s/47f0fb6207ac3f9e2351?file=48907888`, 2024.

[27] Sakina Fatima, Taher A Ghaleb, and Lionel Briand. Flakify: A black-box, language model-based predictor for flaky tests. *IEEE Transactions on Software Engineering*, 2022.

[28] Sakina Fatima, Hadi Hemmati, and Lionel Briand. Flakyfix: Using large language models for predicting flaky test fix categories and test code repair. *IEEE Transactions on Software Engineering*, 2024.

[29] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[30] Michael Fu, Chakkrit Tantithamthavorn, Van Nguyen, and Trung Le. Chatgpt for vulnerability detection, classification, and repair: How far are we? *arXiv preprint arXiv:2310.09810*, 2023.

[31] Vahid Garousi and Barış Küçük. Smells in software test code: A survey of knowledge in industry and academia. *Journal of systems and software*, 138:52–81, 2018.

[32] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.

[33] Cyril Goutte and Eric Gaussier. A probabilistic interpretation of precision, recall and F-score, with implication for evaluation. In *European conference on information retrieval*, pages 345–359. Springer, 2005.

[34] Martin Gruber and Gordon Fraser. A survey on how test flakiness affects developers and what support they need to address it. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 82–92. IEEE, 2022.

[35] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. On calibration of modern neural networks. In *International Conference on Machine Learning*, pages 1321–1330. PMLR, 2017.

[36] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Jian Yin, Daxin Jiang, and M. Zhou. GraphCodeBERT: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

[37] Sarra Habchi, Guillaume Haben, Mike Papadakis, Maxime Cordy, and Yves Le Traon. A qualitative study on the sources, impacts, and mitigation strategies of flaky tests. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 244–255. IEEE, 2022.

[38] Guillaume Haben, Sarra Habchi, Mike Papadakis, Maxime Cordy, and Yves Le Traon. A Replication Study on the Usability of Code Vocabulary in Predicting Flaky Tests. In *18th International Conference on Mining Software Repositories*, 2021.

[39] Negar Hashemi, Amjed Tahir, and Shawn Rasheed. An empirical study of flaky tests in javascript. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 24–34. IEEE, 2022.

[40] Anfeng He, Chong Luo, Xinmei Tian, and Wenjun Zeng. A twofold siamese network for real-time object tracking. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4834–4843, 2018.

[41] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput*, 9(8):1735–1780, 1997.

[42] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. *Applied logistic regression*. John Wiley & Sons, 2013.

[43] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*, 2018.

[44] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[45] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. TreeBERT: A tree-based pre-trained model for programming language. *arXiv preprint arXiv:2105.12485*, 2021.

[46] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, pages 5110–5121. PMLR, 2020.

[47] Jan Keim, Angelika Kaplan, Anne Koziolek, and Mehdi Mirakhorli. Does BERT Understand Code?–An Exploratory Study on the Detection of Architectural Tactics in Code. In *European Conference on Software Architecture*, pages 220–228. Springer, 2020.

[48] G Koch, R Zemel, and R Salakhutdinov. Siamese neural networks for one-shot image recognition. in icml deep learning workshop (vol. 2). 2015.

[49] Emily Kowalczyk, Karan Nair, Zebao Gao, Leo Silberstein, Teng Long, and Atif Memon. Modeling and ranking flaky tests at Apple. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 110–119. IEEE, 2020.

[50] Wing Lam. International Dataset of Flaky Tests (IDoFT), 2020.

[51] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. Root causing flaky tests in a large-scale industrial setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 101–111, 2019.

[52] Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. A study on the lifecycle of flaky tests. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1471–1482, 2020.

[53] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. idflakies: A framework for detecting and partially classifying flaky tests. In *2019 12th ieee conference on software testing, validation and verification (icst)*, pages 312–322. IEEE, 2019.

[54] Wing Lam, August Shi, Reed Oei, Sai Zhang, Michael D Ernst, and Tao Xie. Dependent-test-aware regression testing techniques. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 298–311, 2020.

[55] Wing Lam, Stefan Winter, Angello Astorga, Victoria Stodden, and Darko Marinov. Understanding reproducibility and characteristics of flaky tests through test reruns in java projects. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 403–413. IEEE, 2020.

[56] Wing Lam, Stefan Winter, Anjiang Wei, Tao Xie, Darko Marinov, and Jonathan Bell. A large-scale longitudinal study of flaky tests. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.

[57] Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. Improving chatgpt prompt for code generation. *arXiv preprint arXiv:2305.08360*, 2023.

[58] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for gpt-3? *arXiv preprint arXiv:2101.06804*, 2021.

[59] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[60] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 643–653, 2014.

[61] Danilo Mandic and Jonathon Chambers. *Recurrent neural networks for prediction: learning algorithms, architectures and stability*. Wiley, 2001.

[62] Ehsan Mashhadi and Hadi Hemmati. Applying codebert for automated program repair of java simple bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 505–509. IEEE, 2021.

[63] Mary L McHugh. Interrater reliability: the kappa statistic. *Biochemia medica*, 22(3):276–282, 2012.

[64] Mary L McHugh. The chi-square test of independence. *Biochemia medica*, 23(2):143–149, 2013.

[65] Gledson Melotti, Cristiano Premebida, Jordan J Bird, Diego R Faria, and N Gonçalves. Probabilistic Object Classification using CNN ML-MAP layers. *arXiv preprint arXiv:2005.14565*, 2020.

[66] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming Google-scale continuous testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 233–242, 2017.

[67] Atif Memon and John Micco. How flaky tests in continuous integration. `https://www.youtube.com/watch?v=CrzpkF1-VsA`, 2016.

[68] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[69] J Micco. Flaky tests at google and how we mitigate them—googblogs. com, 2016.

[70] John Micco. Flaky tests at Google and how we mitigate them. `https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html`, 2016.

[71] John Micco. Advances in continuous integration testing at Google. `https://research.google/pubs/pub46593`, 2018.

[72] Christopher Z Mooney, Robert D Duval, and Robert Duvall. *Bootstrapping: A nonparametric approach to statistical inference*. Number 95. sage, 1993.

[73] Jonas Mueller and Aditya Thyagarajan. Siamese recurrent architectures for learning sentence similarity. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[74] David Nadeau and Satoshi Sekine. A survey of named entity recognition and classification. *Lingvisticae Investigationes*, 30(1):3–26, 2007.

[75] Paul Neculoiu, Maarten Versteegh, and Mihai Rotaru. Learning text similarity with siamese recurrent networks. In *Proceedings of the 1st Workshop on Representation Learning for NLP*, pages 148–157, 2016.

[76] Nhan Nguyen and Sarah Nadi. An empirical evaluation of github copilot's code suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 1–5, 2022.

[77] Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. An empirical comparison of pre-trained models of source code. *arXiv preprint arXiv:2302.04026*, 2023.

[78] Robert E Noonan. An algorithm for generating abstract syntax trees. *Computer Languages*, 10(3-4):225–236, 1985.

[79] OpenAI. Openai platform documentation: Continuous model upgrades. `https://platform.openai.com/docs/models/continuous-model-upgrades`, Accessed: 2023.

[80] R OpenAI. Gpt-4 technical report. arxiv 2303.08774. *View in Article*, 2, 2023.

[81] Roy Osherove. *The Art of Unit Testing: with examples in C*. Simon and Schuster, 2013.

[82] Cong Pan, Minyan Lu, and Biao Xu. An Empirical Study on Software Defect Prediction Using CodeBERT Model. *Applied Sciences*, 11(11):4793, 2021.

[83] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J Hellendoorn. Revisiting test smells in automatically generated tests: limitations, pitfalls, and opportunities. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 523–533. IEEE, 2020.

[84] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[85] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(1):1–74, 2021.

[86] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. Surveying the developer experience of flaky tests. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2022.

[87] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. What do developer-repaired flaky tests tell us about the effectiveness of automated flaky test detection? In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, pages 160–164, 2022.

[88] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318. Pmlr, 2013.

[89] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356. IEEE, 2023.

[90] Yu Pei, Jeongju Sohn, Sarra Habchi, and Mike Papadakis. Traf: Time-based repair for asynchronous wait flaky tests in web testing. *arXiv preprint arXiv:2305.08592*, 2023.

[91] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. Tsdetect: An open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1650–1654, 2020.

[92] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

[93] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. What is the vocabulary of flaky tests? In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 492–502, 2020.

[94] Valeria Pontillo, Fabio Palomba, and Filomena Ferrucci. Toward static test flakiness prediction: a feasibility study. In *Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution*, pages 19–24, 2021.

[95] Julian Aron Prenner, Hlib Babii, and Romain Robbes. Can openai's codex fix bugs? an evaluation on quixbugs. In *Proceedings of the Third International Workshop on Automated Program Repair*, pages 69–75, 2022.

[96] Shuofei Qiao, Yixin Ou, Ningyu Zhang, Xiang Chen, Yunzhi Yao, Shumin Deng, Chuanqi Tan, Fei Huang, and Huajun Chen. Reasoning with language model prompting: A survey. *arXiv preprint arXiv:2212.09597*, 2022.

[97] Michel Raymond and François Rousset. An exact test for population differentiation. *Evolution*, pages 1280–1283, 1995.

[98] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[99] Francisco Ribeiro, José Nuno Castro de Macedo, Kanae Tsushima, Rui Abreu, and João Saraiva. Gpt-3-powered type error debugging: Investigating the use of large language models for code repair. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering*, pages 111–124, 2023.

[100] Eric Sven Ristad and Peter N Yianilos. Learning string-edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):522–532, 1998.

[101] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.

[102] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

[103] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. ifixflakies: A framework for automatically fixing order-dependent flaky tests. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 545–555, 2019.

[104] Connor Shorten, Taghi M Khoshgoftaar, and Borko Furht. Text data augmentation for deep learning. *Journal of big Data*, 8:1–34, 2021.

[105] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[106] Chen Sun, Austin Myers, Carl Vondrick, Kevin Murphy, and Cordelia Schmid. VideoBERT: A joint model for video and language representation learning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7464–7473, 2019.

[107] Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. How to fine-tune BERT for text classification? In *China National Conference on Chinese Computational Linguistics*, pages 194–206. Springer, 2019.

[108] Ferdian Thung, David Lo, and Lingxiao Jiang. Automatic defect categorization. In *2012 19th working conference on reverse engineering*, pages 205–214. IEEE, 2012.

[109] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95. Citeseer, 2001.

[110] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[111] Tim Verdonck, Bart Baesens, María Óskarsdóttir, and Seppe vanden Broucke. Special issue on feature engineering editorial. *Machine Learning*, pages 1–12, 2021.

[112] Tássio Virgínio, Luana Martins, Larissa Rocha, Railana Santana, Adriana Cruz, Heitor Costa, and Ivan Machado. JNose: Java Test Smell Detector. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*, pages 564–569, 2020.

[113] Ruixin Wang, Yang Chen, and Wing Lam. ipflakies: A framework for detecting and fixing python order-dependent flaky tests. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 120–124, 2022.

[114] Yaqing Wang, Quanming Yao, James T Kwok, and Lionel M Ni. Generalizing from a few examples: A survey on few-shot learning. *ACM computing surveys (csur)*, 53(3):1–34, 2020.

[115] Anjiang Wei, Pu Yi, Tao Xie, Darko Marinov, and Wing Lam. Probabilistic and systematic coverage of consecutive test-method pairs for detecting order-dependent flaky tests. In *Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings, Part I 27*, pages 270–287. Springer, 2021.

[116] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. *arXiv preprint arXiv:2303.07839*, 2023.

[117] What Makes In-Context Learning Work. Rethinking the role of demonstrations: What makes in-context learning work?

[118] Jiajie Wu. Literature review on vulnerability detection using NLP technology. *arXiv preprint arXiv:2104.11230*, 2021.

[119] Yonghui Wu, Mike Schuster, Z. Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason R. Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Gregory S. Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[120] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*, 2023.

[121] Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 959–971, 2022.

[122] Hu Xu, Bing Liu, Lei Shu, and Philip S Yu. BERT post-training for review reading comprehension and aspect-based sentiment analysis. *arXiv preprint arXiv:1904.02232*, 2019.

[123] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. XLNet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems*, 32, 2019.

[124] Zhewei Yao, Amir Gholami, Sheng Shen, Mustafa Mustafa, Kurt Keutzer, and Michael Mahoney. Adahessian: An adaptive second order optimizer for machine learning. In *proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 10665–10673, 2021.

[125] Zhewei Yao, Amir Gholami, Sheng Shen, Mustafa Mustafa, Kurt Keutzer, and Michael W Mahoney. ADAHESSIAN: An adaptive second order optimizer for machine learning. *arXiv preprint arXiv:2006.00719*, 2020.

[126] Quanjun Zhang, Tongke Zhang, Juan Zhai, Chunrong Fang, Bowen Yu, Weisong Sun, and Zhenyu Chen. A critical review of large language model on software engineering: An example from chatgpt and automated program repair. *arXiv preprint arXiv:2310.08879*, 2023.

[127] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

[128] Wei Zheng, Guoliang Liu, Manqing Zhang, Xiang Chen, and Wenqiao Zhao. Research progress of flaky tests. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 639–646. IEEE, 2021.

[129] Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372*, 2023.

[130] Celal Ziftci and Diego Cavalcanti. De-flake your tests: Automatically locating root causes of flaky tests in code at google. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 736–745. IEEE, 2020.

[131] Behrouz Zolfaghari, Reza M Parizi, Gautam Srivastava, and Yoseph Hailemariam. Root causing, detecting, and fixing flaky tests: State of the art and future roadmap. *Software: Practice and Experience*, 51(5):851–867, 2021.